

Specifying Concurrent Problems: Beyond Linearizability

Armando Castañeda[†] Sergio Rajsbaum[†] Michel Raynal^{*,°}

[†] Instituto de Matemáticas, UNAM, México D.F, 04510, México

^{*} Institut Universitaire de France

[°] IRISA, Université de Rennes 35042 Rennes Cedex, France

armando.castaneda@im.unam.mx rajsbaum@im.unam.mx raynal@irisa.fr

Abstract

Tasks and objects are two predominant ways of specifying distributed problems. A *task* is specified by an input/output relation, defining for each set of processes that may run concurrently, and each assignment of inputs to the processes in the set, the valid outputs of the processes. An *object* is specified by an automaton describing the outputs the object may produce when it is accessed sequentially. Thus, tasks explicitly state what may happen only when sets of processes run concurrently, while objects only specify what happens when processes access the object sequentially. Each one requires its own *implementation* notion, to tell when an execution satisfies the specification. For objects *linearizability* is commonly used, a very elegant and useful consistency condition. For tasks implementation notions are less explored.

These two orthogonal approaches are central, the former in distributed computability, and the later in concurrent programming, yet they have not been unified. Sequential specifications are very convenient, especially important is the *locality* property of linearizability, which states that one can build systems in a modular way, considering object implementations in isolation. However, many important distributed computing problems, including some well-known tasks, have no sequential specification. Also, tasks are one-shot problems with a semantics that is not fully understood (as we argue here), and with no clear locality property, while objects can be invoked in general several times by the same process.

The paper introduces the notion of *interval-sequential* object. The corresponding implementation notion of *interval-linearizability* generalizes linearizability, and allows to associate states along the interval of execution of an operation. Interval-linearizability allows to specify any task, however, there are sequential one-shot objects that cannot be expressed as tasks, under the simplest interpretation of a task. It also shows that a natural extension of the notion of a task is expressive enough to specify any interval-sequential object.

Thus, on the one hand, interval-sequential linearizability explains in more detail the semantics of a task, gives a more precise implementation notion, and brings a locality property to tasks. On the other hand, tasks provide a static specification for automata-based formalisms.

Keywords: asynchronous system, concurrent object, distributed task, linearizability, object composability, sequential specification.

1 Introduction

Concurrent objects and linearizability Distributed computer scientists excel at thinking concurrently, and building large distributed programs that work under difficult conditions with highly asynchronous processes that may fail. Yet, they evade thinking about *concurrent* problem specifications. A central paradigm is that of a shared object that processes may access concurrently [28, 42, 46], but the object is specified in terms of a sequential specification, i.e., an automaton describing the outputs the object produces only when it is accessed sequentially. Thus, a concurrent algorithm seeks to emulate an allowed sequential behavior.

There are various ways of defining what it means for an algorithm to *implement* an object, namely, that it satisfies its sequential specification. One of the most popular consistency conditions is *linearizability* [31], (see surveys [13, 41]). Given a sequential specification of an object, an algorithm *implements* the object if every execution can be transformed to a sequential one such that (1) it respects the real-time order of invocation and responses and (2) the sequential execution is recognized by the automaton specifying the object. It is then said that the corresponding object implementation is *linearizable*. Thus, an execution is linearizable if, for each operation call, it is possible to find a unique point in the interval of real-time defined by the invocation and response of the operation, and these *linearization points* induce a valid sequential execution. Linearizability is very popular to design components of large systems because it is *local*, namely, one can consider linearizable object implementations in isolation and *compose* them without sacrificing linearizability of the whole system [16]. Also, linearizability is a *non-blocking* property, which means that a pending invocation (of a total operation) is never required to wait for another pending invocation to complete. Textbooks such as [6, 28, 42, 46] include more detailed discussions of linearizability.

Linearizability has various desirable properties, additionally to being local and non-blocking: it allows talking about the state of an object, interactions among operations is captured by side-effects on object states; documentation size of an object is linear in the number of operations; new operations can be added without changing descriptions of old operations. However, as we argue here, linearizability is sometimes too restrictive. First, there are problems which have no sequential specifications (more on this below). Second, some problems are more naturally and succinctly defined in term of concurrent behaviors. Third, as is well known, the specification of a problem should be as general as possible, to allow maximum flexibility to both programmers and program executions.

Distributed tasks Another predominant way of specifying a one-shot distributed problem, especially in distributed computability, is through the notion of a *task* [37]. Several tasks have been intensively studied in distributed computability, leading to an understanding of their relative power [27], to the design of simulations between models [8], and to the development of a deep connection between distributed computing and topology [26]. Formally, a task is specified by an input/output relation, defining for each set of processes that may run concurrently, and each assignment of inputs to the processes in the set, the valid outputs of the processes. Implementation notions for tasks are less explored, and they are not as elegant as linearizability. In practice, task and implementation are usually described operationally, somewhat informally. One of the versions widely used is that an algorithm *implements* a task if, in every execution where a set of processes participate (run to completion, and the other crash from the beginning), input and outputs satisfy the task specification.

A main difference between tasks and objects is how they model the concurrency that naturally arises in distributed systems: whiles tasks explicitly state what might happen for several (but no all) concurrency patterns, objects only specify what happens when processes access the object sequentially.

It is remarkable that these two approaches have largely remained independent¹, while the main distributed computing paradigm, *consensus*, is central to both. Neiger [38] noticed this and proposed a generalization of linearizability called *set-linearizability*. He discussed that there are tasks, like *immediate snapshot* [7], with no natural specification as sequential objects. In this task there is a single operation `Immediate_snapshot()`, such that a snapshot of the shared memory occurs immediately after a write. If one wants to model immediate snapshot as an object, the resulting object implements test-and-set, which is contradictory because there are read/write algorithms solving the immediate snapshot task and it is well-known that there are no read/write linearizable implementations of test-and-set. Thus, it is meaningless to ask if there is a linearizable implementation of immediate snapshot because there is no natural sequential specification of it. Therefore, Neiger proposed the notion of a *set-sequential* object, that allows a set of processes to access an object simultaneously. Then, one can define an immediate snapshot set-sequential object, and there are *set-linearizables* implementations.

Contributions We propose the notion of an *interval-sequential* concurrent object, a framework in which an object is specified by an automaton that can express any concurrency pattern of overlapping invocations of operations,

¹Also both approaches were proposed the same year, 1987, and both are seminal to their respective research areas [30, 37].

that might occur in an execution (although one is not forced to describe all of them). The automaton is a direct generalization of the automaton of a sequential object, except that transitions are labeled with sets of invocations and responses, allowing operations to span several consecutive transitions. The corresponding implementation notion of *interval-linearizability* generalizes linearizability and set-linearizability, and allows to associate states along the interval of execution of an operation. While linearizing an execution requires finding linearization *points*, in interval-linearizability one needs to identify a linearization *interval* for each operation (the intervals might overlap). Remarkably, this general notion remains local and non-blocking. We show that most important tasks (including *set agreement* [11]) have no specification neither as a sequential objects nor as a set-sequential objects, but they can be naturally expressed as interval-sequential objects.

Establishing the relationship between tasks and (sequential, set-sequential and interval-sequential) automata-based specifications is subtle, because tasks admit several natural interpretations. Interval-linearizability is a framework that allows to specify any task, however, there are sequential one-shot objects that cannot be expressed as tasks, under the simplest interpretation of a task. Hence, interval-sequential objects have strictly more power to specify one-shot problems than tasks. However, a natural extension of the notion of a task has the same expressive power to specify one-shot concurrent problems, hence strictly more than sequential and set-sequential objects. See Figure 1. Interval-linearizability goes beyond unifying sequentially specified objects and tasks, it sheds new light on both of them. On the one hand, interval-sequential linearizability provides an explicit operational semantics to a task (whose semantics, as we argue here, is not well understood), gives a more precise implementation notion, and brings a locality property to tasks. On the other hand, tasks provide a static specification for automata-based formalisms such as sequential, set-sequential and interval-sequential objects.

Related work Many consistency conditions have been proposed to define the correct behavior of sequentially specified objects, that guarantee that all the processes see the same sequence of operations applied to the object. Among the most notable are *atomicity* [34, 35, 36], *sequential consistency* [33], and *linearizability* [31]. (See surveys [13, 41], and textbooks such as [6, 28, 42, 43])². An extension of linearizability suited to relativistic distributed systems is presented in [22]. *Normality* consistency [21] can be seen as an extension of linearizability to the case where an operation can involve more than one object.

Neiger proposed unifying sequential objects and tasks, and defined set-linearizability [38]. In the automaton specifying a set-sequential object, transitions between states involve more than one operation; these operations are allowed to occur concurrently and their results can be concurrency-dependent. Thus, linearizability corresponds to the case when the transitions always involve a single operation. Later on it was again observed that for some concurrent objects it is impossible to provide a sequential specification, and similar notion, but based on histories, was proposed [25] (no properties were proved). Transforming the question of wait-free read/write solvability of a one-shot sequential object, into the question of solvability of a task was suggested in [18]. The extension of tasks we propose here is reminiscent to the construction in [18].

Higher dimensional automata are used to model execution of concurrent operations, and are the most expressive model among other common operations [19]. They can model transitions which consists of sets of operations, and hence are related to set-linearizability, but do not naturally model interval-linearizability, and other concerns of concurrent objects. There is work on partial order semantics of programs, including more flexible notions of linearizability, relating two arbitrary sets of histories [15].

Roadmap The paper is composed of 6 sections. It considers that the basic definitions related to linearizability are known. First, Section 2 uses a simple example to illustrate the limitations of both linearizability and set-linearizability. Then, Section 3 introduces the notion of an interval-sequential concurrent object, which makes it possible to specify the correct concurrent patterns, without restricting them to be sequential patterns. Section 4 defines interval-linearizability and shows it is local and non-blocking. Then, Section 5 compares the ability of tasks and interval-sequential objects to specify one-shot problems. Finally, Section 6 concludes the paper.

²Weaker consistency conditions such as *causal consistency* [3], *lazy release consistency* [32], or *eventual consistency* [47] are not addressed here.

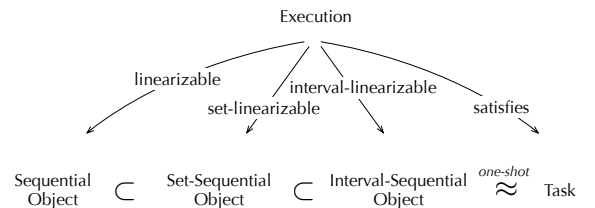


Figure 1: Objects and consistency conditions. The equivalence is between *refined* tasks and one-shot interval-sequential objects.

2 Limitations of linearizability and set-linearizability

Here we discuss in more detail limitations of sequential and set-sequential specifications (linearizability and set-linearizability). As a running example we use write-snapshot, a natural task that is implementable from read/write registers and has no natural specification as a sequential or set-sequential object. Many other tasks have the same problems. Appendix C presents other examples and additional details.

2.1 The write-snapshot task

Definition and implementation of write-snapshot Sometimes we work with objects with two operations, but that are intended to be used as one. For instance, a snapshot object [1] has operations `write()` (sometimes called `update`) and `snapshot()`. This object has a sequential specification and there are linearizable read/write algorithms implementing it (see, e.g., [6, 28, 42, 46]). But many times, a snapshot object is used in a canonical way, namely, each time a process invokes `write()`, immediately after it always invokes `snapshot()`. Indeed, one would like to think of such an object as providing a single operation, `write_snapshot()`, invoked with a value x to be deposited in the object, and when the operation returns, it gives back to the invoking process a snapshot of the contents of the object. It turns out that this write-snapshot object has neither a natural sequential nor a set-sequential specification. However, it can be specified as a task and actually is implementable from read/write registers.

In the *write-snapshot* task, each process p_i starts with a private input v_i and outputs a set set_i satisfying the following:

- Self-inclusion: $\langle i, v_i \rangle \in set_i$.
- Containment: $\forall i, j : (set_i \subseteq set_j) \vee (set_j \subseteq set_i)$.

Note that the specification of write-snapshot is highly concurrent: it only states what processes might decide when they run until completion, regardless of the specific interleaving pattern of invocations and responses. A simple write-snapshot algorithm based on read/write registers, is in Figure 2 below.

The *immediate snapshot* task [7] is defined by adding an Immediacy requirement to the Self-inclusion and Containment requirements of the write-snapshot task.

- Immediacy: $\forall i, j : [(\langle j, v_j \rangle \in set_i) \wedge (\langle i, v_i \rangle \in set_j)] \Rightarrow (set_i = set_j)$.

Figure 2 contains an algorithm that implements write-snapshot (same idea of the well-known algorithm of [1]). The internal representation of write-snapshot is made up of an array of single-writer multi-reader atomic registers $MEM[1..n]$, initialized to $[\perp, \dots, \perp]$. In the following, to simplify the presentation we suppose that the value written by p_i is i , and the pair $\langle i, v_i \rangle$ is consequently denoted i . When a process p_i invokes `write_snapshot(i)`, it first writes its value i in $MEM[i]$ (line 01). Then p_i issues repeated classical “double collects” until it obtains two successive read of the full array MEM , which provide it with the same set of non- \perp values (lines 02-05). When such a successful double collect occurs, p_i returns the content of its last read of the array MEM (line 06). Let us recall that the reading of the n array entries are done asynchronously and in an arbitrary order. In Appendix B, it is shown that this algorithm implements the write-snapshot task.

<p>operation <code>write_snapshot(i)</code> is % issued by p_i</p> <p>(01) $MEM[i] \leftarrow i$;</p> <p>(02) $new_i \leftarrow \cup_{1 \leq j \leq n} \{MEM[j] \text{ such that } MEM[j] \neq \perp\}$;</p> <p>(03) repeat $old_i \leftarrow new_i$;</p> <p>(04) $new_i \leftarrow \cup_{1 \leq j \leq n} \{MEM[j] \text{ such that } MEM[j] \neq \perp\}$</p> <p>(05) until $(old_i = new_i)$ end repeat;</p> <p>(06) return (new_i).</p>
--

Figure 2: A write-snapshot algorithm

Can the write-snapshot task be specified as a sequential object? Suppose there is a deterministic sequential specification of write-snapshot. Since the write-snapshot task is implementable from read/write registers, one expects that there is a linearizable algorithm A implementing the write-snapshot task from read/write registers. But A is linearizable, hence any of its executions can be seen as if all invocations occurred one after the other, in some order. Thus, always there is a first invocation, which must output the set containing only its input value. Clearly, using A as a building block, one can trivially solve test-and-set. This contradicts the fact that test-and-set

cannot be implemented from read/write registers. The contradiction comes from the fact that, in a deterministic sequential specification of write-snapshot, the values in the output set of a process can only contain input values of operations that happened before. Such a specification is actually modelling a proper subset of all possible relations between inputs and outputs, of the distributed problem we wanted to model at first. This phenomenon is more evident when we consider the execution in Figure 3, which can be produced by the write-snapshot algorithm in Figure 2 in the Appendix.

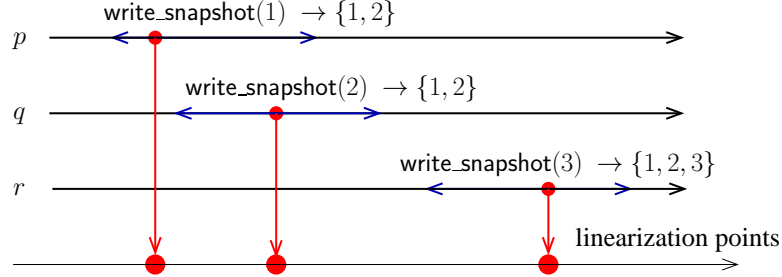


Figure 3: A linearizable write-snapshot execution that predicts the future

Consider a non-deterministic sequential specification of write-snapshot (the automaton is in Appendix B). When linearizing the execution in Figure 3, one has to put either the invocation of p or q first, in either case the resulting sequential execution seems to say that the first process predicted the future and knew that q will invoke the task. The linearization points in the figure describe a possible sequential ordering of operations. These anomalous future-predicting sequential specifications result in linearizations points without the intended meaning of “as if the operation was atomically executed at that point.”

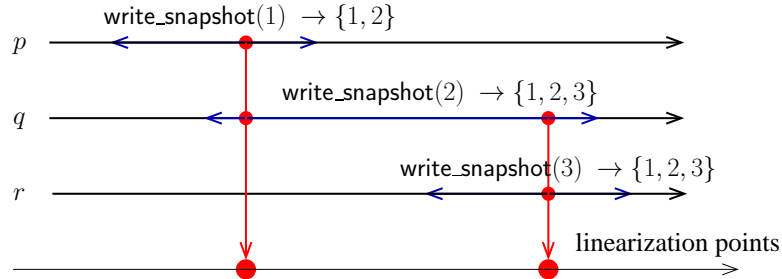


Figure 4: A write-snapshot execution that is not set-linearizable

Why set-linearizability is not enough Neiger noted the problems with the execution in Figure 3 discussed above, in the context of the immediate snapshot task. He proposed in [38] the idea that a specification should allow to express that sets of operations that can be concurrent. He called this notion *set-linearizability*. In set-linearizability, an execution accepted by a *set-sequential* automaton is a sequence of non-empty sets with operations, and each set denotes operations that are executed concurrently. In this way, in the execution in Figure 3, the operations of p and q would be set-linearized together, and then the operation of r would be set-linearized alone at the end. While set-linearizability is sufficient to model the immediate-snapshot task, it is not enough for specifying most other tasks.

Consider the write-snapshot task. In set-linearizability, in the execution in Figure 4 (which can be produced by the write-snapshot algorithm, but is not a legal immediate snapshot execution), one has to decide if the operation of q goes together with the one of p or r . In either case, in the resulting execution a process seems to predict a future operation. In this case the problem is that there are operations that are affected by several operations that are not concurrent (in Figure 4, q is affected by both p and r , whose operations are not concurrent). This cannot be expressed as a set-sequential execution. Hence, to succinctly express this type of behavior, we need a more flexible framework in which it is possible to express that an operation happens in an interval of time that can be affected by several operations.

2.2 Additional examples of tasks with no sequential specification and a potential solution

As we shall see, most tasks are problematic for dealing with them through linearizability, and have no deterministic sequential specifications. Some have been studied in the past, such as the following, discussed in more detail in Appendix C.1.

- *adopt-commit* [17] is a one-shot shared-memory object useful to implement round-based protocols for set-agreement and consensus. Given an input u to the object, the result is an output of the form $(commit, v)$ or $(adopt, v)$, where *commit/adopt* is a decision that indicates whether the process should decide value v immediately or adopt it as its preferred value in later rounds of the protocol.
- *conflict detection* [4] has been shown to be equivalent to the adopt-commit. Roughly, if at least two different values are proposed concurrently at least one process outputs true.
- *safe-consensus* [2], a weakening of consensus, where the agreement condition of consensus is retained, but the validity condition becomes: if the first process to invoke it returns before any other process invokes it, then it outputs its input; otherwise the consensus output can be arbitrary, not even the input of any process.
- *immediate snapshot* [7], which plays an important role in distributed computability [5, 7, 45]. A process can write a value to the shared memory using this operation, and gets back a snapshot of the shared memory, such that the snapshot occurs immediately after the write.
- *k-set agreement* [11], where processes agree on at most k of their input values.
- *Exchanger* [25], is a Java object that serves as a synchronization point at which threads can pair up and atomically swap elements.

Splitting an operation in two To deal with these problematic tasks, one is tempted to separate an operation into two operations, **set** and **get**. The first communicates the input value of a process, while the second produces an output value to a process. For instance, k -set agreement is easily transformed into an object with a sequential specification, simply by accessing it through **set** to deposit a value into the object and **get** that returns one of the values in the object. In fact, every task can be represented as a sequential object by splitting the operation of the task in two operations (proof in Appendix C.2).

Separating an operation into a proposal operation and a returning operation has several problems. First, the program is forced to produce two operations, and wait for two responses. There is a consequent loss of clarity in the code of the program, in addition to a loss in performance, incurred by a two-round trip delay. Also, the intended meaning of linearization points is lost; an operation is now linearized at *two* linearization points. Furthermore, the resulting object may provably *not* be the same; a phenomenon that has been observed several times in the context of iterated models (e.g., in [12, 20, 40]) is that the power of the object can be increased, if one is allowed to invoke another object in between the two operations. Further discussion of this issue is in Appendix C.2.

3 Concurrent Objects

This section defines the notion of an *interval-sequential* concurrent object, which allows to specify behaviors of all the valid concurrent operation patterns. These objects include as special cases sequential and set-sequential objects. To this end, the section also describes the underlying computation model.

3.1 System model

The system consists of n asynchronous sequential processes, $P = \{p_1, \dots, p_n\}$, which communicate through a set of concurrent objects, OBS . Each consistency condition specifies the behaviour of an object differently, for now we only need to define its interface, which is common to all conditions. The presentation follows [9, 31, 42].

Given a set OP of operations offered by the objects of the system to the processes P , let Inv be the set of all invocations to operations that can be issued by a process in a system, and Res be the set of all responses to the invocations in Inv . There are functions

$$\begin{aligned}
 id &: Inv \rightarrow P \\
 op &: Inv \rightarrow OP \\
 op &: Res \rightarrow OP \\
 res &: Res \rightarrow Inv \\
 obj &: OP \rightarrow OBS
 \end{aligned} \tag{1}$$

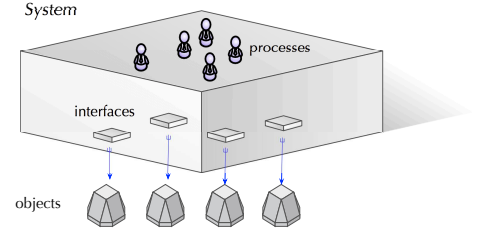
where $id(in)$ tells which process invoked $in \in Inv$, $op(in)$ tells which operation was invoked, $op(r)$ tells which operation was responded, $res(r)$ tells which invocation corresponds to $r \in Res$, and $obj(oper)$ indicates the object that offers operation $oper$. There is an induced function $id : Res \rightarrow P$ defined by $id(r) = id(res(r))$. Also, induced functions $obj : Inv \rightarrow OBS$ defined by $obj(in) = obj(op(in))$, and $obj : Res \rightarrow OBS$ defined

by $obj(r) = obj(op(r))$. The set of operations of an object X , $OP(X)$, consists of all operations $oper$, with $obj(oper) = X$. Similarly, $Inv(X)$ and $Res(X)$ are resp. the set of invocations and responses of X .

A *process* is a deterministic automaton that interacts with the objects in OBS . It produces a sequence of steps, where a *step* is an invocation of an object's operation, or reacting to an object's response (including local processing). Consider the set of all operations OP of objects in OBS , and all the corresponding possible invocations Inv and responses Res . A process p is an automaton (Σ, ν, τ) , with states Σ and functions ν, τ that describe the interaction of the process with the objects. Often there is also a set of initial states $\Sigma_0 \subseteq \Sigma$. Intuitively, if p is in state σ and $\nu(\sigma) = (op, X)$ then in its next step p will apply operation op to object X . Based on its current state, X will return a response r to p and will enter a new state, in accordance to its transition relation. Finally, p will enter state $\tau(\sigma, r)$ as a result of the response it received from X .

Finally, a *system* consists of a set of processes, P , a set of objects OBS so that each $p \in P$ uses a subset of OBS , together with an initial state for each of the objects.

A *configuration* is a tuple consisting of the state of each process and each object, and a configuration is *initial* if each process and each object is in an initial state. An *execution* of the system is modelled by a sequence of events H arranged in a total order $\hat{H} = (H, <_H)$, where each event is an invocation $in \in Inv$ or a response $r \in Res$, that can be produced following the process automata, interacting with the objects. Namely, an execution starts, given any initial configuration, by having any process invoke an operation, according to its transition relation. In general, once a configuration is reached, the next event can be a response from an object to an operation of a process or an invocation of an operation by a process whose last invocation has been responded. Thus, an execution is well-formed, in the sense that it consists of an interleaving of invocations and responses to operations, where a processes invokes an operation only when its last invocation has been responded.



3.2 The notion of an *Interval-sequential object*

To generalize the usual notion of a sequential object e.g. [9, 31] (recalled in Appendix A), instead of considering sequences of invocations and responses, we consider sequences of *sets* of invocations and responses. An *invoking concurrency class* $C \subseteq 2^{Inv}$, is a non-empty subset of Inv such that C contains at most one invocation by the same process. A *responding concurrency class* $C, C \subseteq 2^{Res}$, is defined similarly.

Interval-sequential execution An *interval-sequential execution* h is an alternating sequence of invoking and responding concurrency classes, starting in an invoking class, $h = I_0, R_0, I_1, R_1, \dots, I_m, R_m$, where the following conditions are satisfied

1. For each $I_i \in h$, any two invocations $in_1, in_2 \in I_i$ are by different processes, $id(in_1) \neq id(in_2)$. Similarly, for $R_i \in h$ if $r_1, r_2 \in R_i$ then $id(r_1) \neq id(r_2)$,
2. Let $r \in R_i$ for some $R_i \in h$. Then there is $in \in I_j$ for some $j \leq i$, such that $res(r) = in$ and furthermore, there is no other in' with $id(in) = id(in')$ with $in' \in I_{j'}, j < j' \leq i$.

It follows that an execution h consists of matching invocations and responses, perhaps with some pending invocations with no response.

Interval-sequential object An *interval-sequential object* X is a (not necessarily finite) Mealy state machine $(Q, 2^{Inv(X)}, 2^{Res(X)}, \delta)$ whose output values R are responding concurrency classes R of X , $R \subseteq 2^{Res(X)}$, are determined both by its current state $s \in Q$ and the current input $I \in 2^{Inv(X)}$, where I is an invoking concurrency class of X . There is a set of *initial states* Q_0 of X , $Q_0 \subseteq Q$. The transition relation $\delta \subseteq Q \times 2^{Inv(X)} \times 2^{Res(X)} \times Q$ specifies both, the output of the automaton and its next state. If X is in state q and it receives as input a set of invocations I , then, if $(R, q') \in \delta(q, I)$, the meaning is that X may return the non-empty set of responses R and move to state q' . We stress that always both I and R are non-empty sets.

Interval-sequential execution of an object Consider an initial state $q_0 \in Q_0$ of X and a sequence of inputs I_0, I_1, \dots, I_m . Then a sequence of outputs that X may produce is R_0, R_1, \dots, R_m , where $(R_i, q_{i+1}) \in \delta(q_i, I_i)$. Then the *interval-sequential execution* of X starting in q_0 is $q_0, I_0, R_0, q_1, I_1, R_1, \dots, q_m, I_m, R_m$. However, we require that the object's response at a state uniquely determines the new state, i.e. we assume if $\delta(q, I_i)$ contains (R_i, q_{i+1}) and (R_i, q'_{i+1}) then $q_{i+1} = q'_{i+1}$. Then we may denote the interval-sequential execution of X , starting in q_0 by $h = I_0, R_0, I_1, R_1, \dots, I_m, R_m$, because the sequence of states q_0, q_1, \dots, q_m is uniquely determined by

q_0 , and by the sequences of inputs and responses. When we omit mentioning q_0 we assume there is some initial state in Q_0 that can produce h .

Notice that X may be non-deterministic, in a given state q_i with input I_i it may move to more than one state and return more than one response. Also, sometimes it is convenient to require that the object is *total*, meaning that, for every singleton set $I \in 2^{Inv}$ and every state q in which the invocation inv in I is not pending, there is an $(R, q') \in \delta(q, I)$ in which there is a response to inv in R .

Our definition of interval-sequential execution is motivated by the fact that we are interested in *well-formed* executions $h = I_0, R_0, I_1, R_1, \dots, I_m, R_m$. Informally, the processes should behave well, in the sense that a process does not invoke a new operation before its last invocation received a response. Also, the object should behave well, in the sense that it should not return a response to an operation that is not pending.

The *interval-sequential specification* of X , $ISSpec(X)$, is the set of all its interval-sequential executions.

Representation of interval-sequential executions In general, we will be thinking of an interval-sequential execution h as an alternating sequence of invoking and responding concurrency classes starting with an invoking class, $h = I_0, R_0, I_1, R_1, \dots, I_m, R_m$. However, it is sometimes convenient to think of an execution as a total order $\hat{S} = (S, \xrightarrow{S})$ on a subset $S \subseteq CC(X)$, where $CC(X)$, is the set with all invoking and responding concurrency classes of X ; namely, $h = I_0 \xrightarrow{S} R_0 \xrightarrow{S} I_1 \xrightarrow{S} R_1 \xrightarrow{S} \dots \xrightarrow{S} I_m \xrightarrow{S} R_m$.

In addition, the execution $h = I_0, R_0, I_1, R_1, \dots, I_m, R_m$ can be represented by a table, with a column for each element in the sequence h , and a row for each process. A member $in \in I_j$ invoked by p_k (resp. a response $r \in R_j$ to p_k) is placed in the k 'th row, at the $2j$ -th column (resp. $2j + 1$ -th column). Thus, a transition of the automaton will correspond to two consecutive columns, I_j, R_j . See Figure 5, and several more examples in the figures below.

Interval-sequential objects include as particular cases set-sequential and sequential objects, as illustrated in Figure 1.

Remark 1 (Sequential and Set-sequential objects). *Let X be an interval-sequential object, $(Q, 2^{Inv(X)}, 2^{Res(X)}, \delta)$. Suppose for all states q and all I , if $\delta(q, I) = (R, q')$, then $|R| = |I|$, and additionally each $r \in R$ is a response to one $in \in I$. Then X is a set-sequential object. If in addition, $|I| = |R| = 1$, then X is a sequential object in the usual sense.*

3.3 Examples: Validity and validity with abort

Consider an object X with a single operation $validity(x)$, that can be invoked by each process, with a *proposed* input parameter x , and a very simple specification: an operation returns a value that has been proposed. This problem is easily specified as a task, see Appendix D.2. Indeed, many tasks include this property, such as consensus, set-agreement, etc. As an interval-sequential object, it is formally specified by an automaton, where each state q is labeled with two values, $q.vals$ is the set of values that have been proposed so far, and $q.pend$ is the set of processes with pending invocations. The initial state q_0 has $q_0.vals = \emptyset$ and $q_0.pend = \emptyset$. If in is an invocation to the object, let $val(in)$ be the proposed value, and if r is a response from the object, let $val(r)$ be the responded value. For a set of invocations I (resp. responses R) $vals(I)$ denotes the proposed values in I (resp. $vals(R)$). The transition relation $\delta(q, I)$ contains all pairs (R, q') such that:

- If $r \in R$ then $id(r) \in q.pend$ or there is an $in \in I$ with $id(in) = id(r)$,
- If $r \in R$ then $val(r) \in q.vals$ or there is an $in \in I$ with $val(in) = val(r)$, and
- $q'.vals = q.val \cup vals(I)$ and $q'.pend = (q.pend \cup ids(I)) \setminus ids(R)$.

On the right of Figure 5 there is part of a validity object automaton. On the left of Figure 5 is illustrated an interval-sequential execution with the vertical red double-dot lines: I_0, R_0, I_1, R_1 , where $I_0 = \{p.validity(1), q.validity(2)\}$, $R_0 = \{p.resp(2)\}$, $I_1 = \{r.validity(3)\}$, $R_1 = \{q.sfresp(3), r.resp(1)\}$.

The interval-linearizability consistency notion described in Section 4 will formally define how a general execution (blue double-arrows in the figure) can be represented by an interval-sequential execution (red double-dot lines), and hence tell if it satisfies the validity object specification. Notice that the execution in Figure 5 shows that the validity object has no specification neither as a sequential nor as a set-sequential object, for reasons similar to those discussed in Section 2.1 about Figure 4.

Augmenting the validity object with an `abort()` operation As an illustration of the expressiveness of an interval-sequential automaton, let us add an operation denoted `abort()` to the validity object, to design a *validity k -abort*

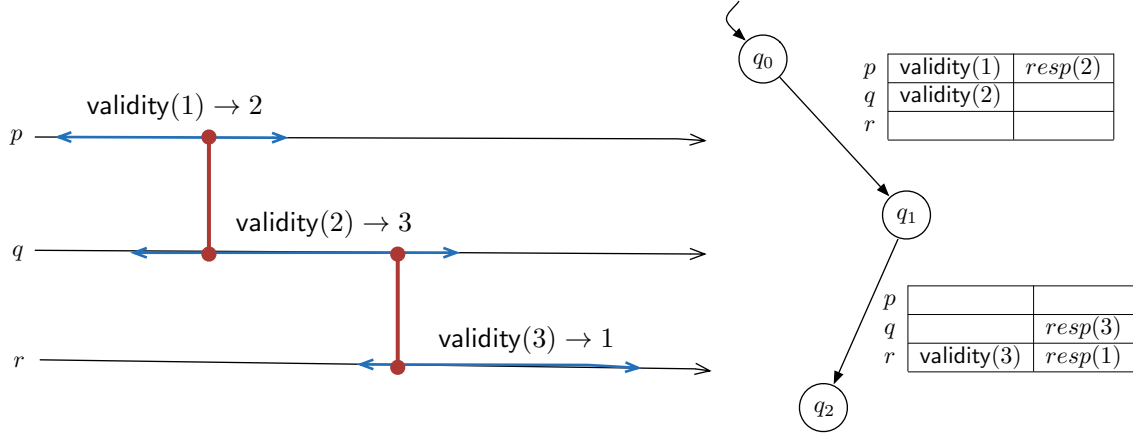


Figure 5: An execution of a validity object, and the corresponding part of an interval-sequential automata

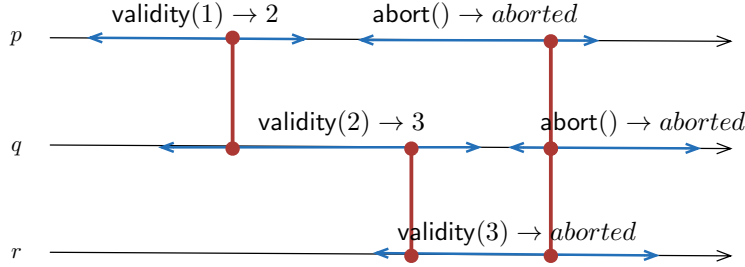


Figure 6: An execution of a Validity-Abort object (1)

object. Since the validity object is not set-linearizable, neither is the validity with abort object. Intuitively, a process can invoke `abort()` to “block” the object, but this might happen only if there are at least k concurrent abort operations. The operation `abort()` returns either *aborted* or *notAborted*, to indicate its result. If all the concurrent `abort()` operations return *aborted*, then any operation happening together or after them, returns *aborted* as well. Hence, if only one process invokes `abort()` then the object behaves as a *Validity* object. How do we formally argue that the execution in Figure 6 is correct? Interval-Linearizability is a correctness implementation notion that serves this purpose, defined next. In Appendix C.3, the validity object is formally defined.

4 Interval-Linearizability

We first define interval-linearizability and then prove it is local and non-blocking.

4.1 The notion of interval-linearizability

Interval-sequential execution of the system Consider a subset $S \subseteq CC$ of the concurrency classes of the objects OBS in the system and an interval-sequential execution $\hat{S} = (S, \xrightarrow{S})$, defining an alternating sequence of invoking and responding concurrency classes, starting with an invoking class. For an object X , the *projection of \hat{S} at X* , $\hat{S}|_X = (S_X, \xrightarrow{S_X})$, is defined as follows: (1) for every $C \in S$ with at least one invocation or response on X , S_X contains a concurrency class C' , consisting of the (non-empty) subset of C of all invocations or responses of X , and (2) for every $C', C'' \in S_X$, $C' \xrightarrow{S_X} C''$ if and only if there are $T', T'' \in S$ such that $C' \subseteq T'$, $C'' \subseteq T''$ and $T' \xrightarrow{S} T''$.

We say that $\hat{S} = (S, \xrightarrow{S})$ is an *interval-sequential execution of the system* if $\hat{S}|_X$ is an interval-sequential execution of X for every $X \in OBS$. That is, if $\hat{S}|_X \in ISSpec(X)$, the interval-sequential specification of X , for every $X \in OBS$. Let $\hat{S} = (S, \xrightarrow{S})$ be an interval-sequential execution. For a process p , the *projection of \hat{S}*

at p , $\widehat{S}|_p = (S_p, \xrightarrow{S_p})$, is defined as follows: (1) for every $C \in S$ with an invocation or response by p , S_p contains a class C with the invocation or response by p (there is at most one event by p in C), and (2) for every $a, b \in S_p$, $a \xrightarrow{S_p} b$ if and only if there are $T', T'' \in S$ such that $a \in T'$, $b \in T''$ and $T' \xrightarrow{S} T''$.

Interval-linearizability Recall that an execution of the system is a sequence of invocations and responses (Section 3.1). An invocation in an execution E is *pending* if it has no matching response, otherwise it is *complete*. An *extension* of an execution E is obtained by appending zero or more responses to pending invocations.

An *operation call* in E is a pair consisting of an invocation and its matching response. Let $\text{comp}(E)$ be the sequence obtained from E by removing its pending invocations. The order in which invocation and responses in E happened, induces the following partial order: $\widehat{OP} = (OP, \xrightarrow{op})$ where OP is the set with all operation calls in E , and for each pair $op_1, op_2 \in OP$, $op_1 \xrightarrow{op} op_2$ if and only if $\text{term}(op_1) < \text{init}(op_2)$ in E , namely, the response of op_1 appears before the invocation of op_2 . Given two operation op_1 and op_2 , op_1 *precedes* op_2 if $op_1 \xrightarrow{op} op_2$, and they are *concurrent* if $op_1 \not\xrightarrow{op} op_2$ and $op_2 \not\xrightarrow{op} op_1$.

Consider an execution of the system E and its associated partial order $\widehat{OP} = (OP, \xrightarrow{op})$, and let $\widehat{S} = (S, \xrightarrow{S})$ be an interval-sequential execution. We say that an operation $a \in OP$ *appears* in a concurrency class $S' \in S$ if its invocation or response is in S' . Abusing notation, we write $a \in S'$. We say that \xrightarrow{S} *respects* \xrightarrow{op} , also written as $\xrightarrow{op} \subseteq \xrightarrow{S}$, if for every $a, b \in OP$ such that $a \xrightarrow{op} b$, for every $T', T'' \in S$ with $a \in T'$ and $b \in T''$, it holds that $T' \xrightarrow{S} T''$.

Definition 1 (Interval-linearizability). *An execution E is interval-linearizable if there is an extension \overline{E} of E and an interval-sequential execution $\widehat{S} = (S, \xrightarrow{S})$ such that*

1. *for every process p , $\text{comp}(\overline{E})|_p = \widehat{S}|_p$,*
2. *for every object X , $\widehat{S}|_X \in \text{ISS}(X)$ and*
3. *\xrightarrow{S} respects \xrightarrow{op} , where $\widehat{OP} = (OP, \xrightarrow{op})$ is the partial order associated to $\text{comp}(\overline{E})$.*

We say that $\widehat{S} = (S, \xrightarrow{S})$ is an *interval-linearization* of E .

Remark 2 (Linearizability and set-linearizability). *When we restrict to interval-sequential executions in which for every invocation there is a response to it in the very next concurrency class, then interval-linearizability boils down to set-linearizability. If in addition we demand that every concurrency class contains only one element, then we have linearizability. See Figure 1.*

We can now complete the example of the validity object. In Figure 7 there is an interval linearization of the execution in Figure 5. Similarly, for the validity with abort object, in Figure 8 there is an interval linearization of the execution in Figure 6.

	<i>init</i>	<i>term</i>	<i>init</i>	<i>term</i>
p	validity(1)	resp(2)		
q	validity(2)			resp(3)
r			validity(3)	resp(1)

Figure 7: An execution of a Validity object

	<i>init</i>	<i>term</i>	<i>init</i>	<i>term</i>	<i>init</i>	<i>term</i>
p	validity(1)	resp(2)			abort()	resp(aborted)
q	validity(2)			resp(3)	abort()	resp(aborted)
r			validity(3)			resp(aborted)

Figure 8: An execution of a Validity-Abort object (2)

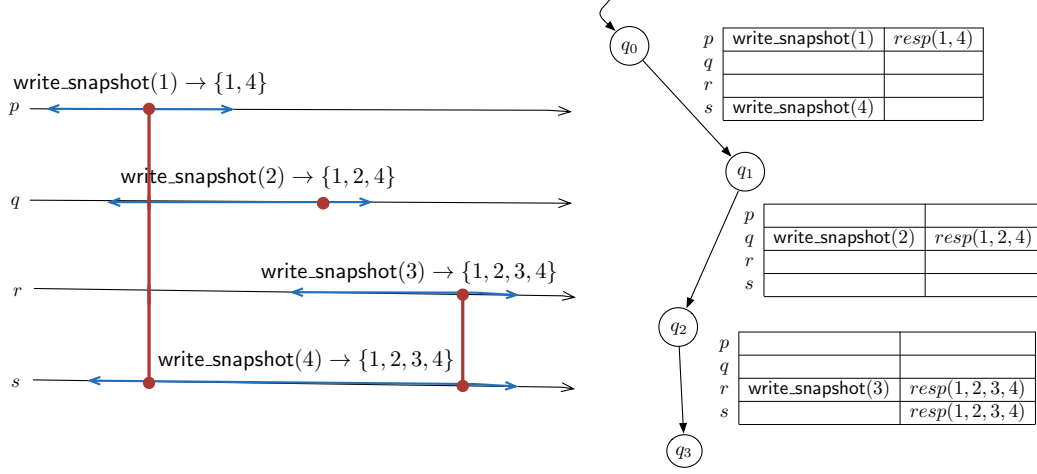


Figure 9: An execution of the write-snapshot task.

4.2 An interval-sequential implementation

Once we have formally defined the notion of interval-linearizability, we can show that the write-snapshot algorithm in Section 2.1 is interval-linearizable.

The write-snapshot interval-sequential object Here is a formal definition of this task, using an interval-sequential object based on the validity object of Section 3.3. The write-snapshot object X has a single operation `write_snapshot(x)` that can be invoked by each process, with a *proposed* input parameter x , and returns a set. In the interval-sequential automata each state q is labeled with two values, $q.vals$ is the set of id-values that have been proposed so far, and $q.pend$ is the set of processes with pending invocations. The initial state q_0 has $q_0.vals = \emptyset$ and $q_0.pend = \emptyset$. If in is an invocation to the object, let $val(in)$ be the proposed value, and $(id(in), val(in))$ be the *proposed id-value pair*. If r is a response from the object, let $val(r)$ be the responded id-value pair. For a set of invocations I (resp. responses R) $vals(I)$ denotes the proposed id-value pairs in I (resp. $vals(R)$). The transition relation $\delta(q, I)$ contains all pairs (R, q') such that:

- If $r \in R$ then $id(r) \in q.pend$ or there is an $in \in I$ with $id(in) = id(r)$,
- If $r \in R$ then $val(r) = q.val \cup vals(I)$
- $q'.vals = q.val \cup vals(I)$ and $q'.pend = (q.pend \cup ids(I)) \setminus ids(R)$.

An example of an execution an the transitions through the automata is in Figure 9.

The write-snapshot algorithm is interval-linearizable The specification of a write-snapshot object contains every interval-sequential execution satisfying the self-containment and containment properties (Appendix B contains a correctness proof in the usual style, without interval-linearizability), thus, to show that an execution of the algorithm is interval-linearizable, we need to transform it into a interval-sequential execution that satisfy the real-time order of invocations and responses.

As with linearizability, interval-linearizability specifies a safety property, it is not about liveness. Thus, before showing that the algorithm of Figure 2 is interval-linearizable, we recall the usual termination arguments for this style of snapshot algorithm. The invocation of `write_snapshot()` by any process p_i terminates, because, as the number of processes is fixed (equal to n), and a process invokes `write_snapshot()` at most once, it follows that a process can execute at most $(n - 1)$ double collects where each time it sees new values.

Theorem 1. *The write-snapshot algorithm of Figure 2 is interval-linearizable.*

Proof The proof is very similar to the usual linearizability proof for the obstruction-free implementation of a snapshot object (we follow [42] (Sect. 8.2.1)), except that now *two* points have to be identified, one for the invocation of an operation and one for the response.

Consider any execution E and let p_i be any process that terminates. As it returns a value set_i (line 06), we have $set_i = old_i = new_i$ where new_i corresponds to the last asynchronous read of $MEM[1..n]$ by p_i , and old_i

corresponds to the previous asynchronous read of $MEM[1..n]$. Let $\tau[old_i]$ the time at which terminates the read of $MEM[1..n]$ returning old_i , and $\tau[new_i]$ the time at which starts the read of $MEM[1..n]$ returning new_i . As $old_i = new_i$, it follows that there is a time τ_i , such that $\tau[old_i] \leq \tau_i \leq \tau[new_i]$ and, due to the termination predicate of line 05, the set of non- \perp values of $MEM[1..n]$ at time τ_i is equal to set_i .

For any process p_i that terminates with set_i , we pick a time τ_i as described above. Let $\bar{\tau} = \tau_{x_1} \leq \tau_{x_2} \leq \dots \leq \tau_{x_m}$ be the ordered sequence of chosen times, assuming the number of processes that terminate is m ($m \leq n$). Clearly if $\tau_i = \tau_j$, then $set_i = set_j$, but it is possible that $set_i = set_j$, with $\tau_i < \tau_j$, in case there is no write in between τ_i and τ_j . Thus, for each longest subsequence of times in $\bar{\tau}$ with the same set set_i , we pick as representative, the first time in the subsequence, and consider the following subsequence $\bar{\tau}'$ of $\bar{\tau}$, where p ($1 \leq p \leq m$) is the number of different sets returned by the processes. The subsequence is $\bar{\tau}' = \tau_{x'_1} < \tau_{x'_2} < \dots < \tau_{x'_p}$, where the sets $set_{x'_1}, set_{x'_2}, \dots, set_{x'_p}$ are all different.

For each subindex x'_i in $\bar{\tau}'$, consider the set that is output $set_{x'_i}$. Let $A_{x'_i}$ be the set of processes in the execution that output $set_{x'_i}$. Using these sets and the sequence of times above, we define an interval-sequential execution as follows. The interval-sequential execution $\hat{S} = (S, \xrightarrow{S})$ consists of an alternating sequence of invoking and responding concurrency classes. The first invoking concurrency class I_1 has all invocations of processes in $set_{x'_1}$, then R_1 , the responding concurrency class with all responses by processes in $A_{x'_1}$, followed by I_2 , the concurrency class with all invocations in $set_{x'_2} \setminus set_{x'_1}$, and the responding class with all responses by processes in $A_{x'_2}$, and so on. For an example, see the interval sequential execution in the right of Figure 9 in Appendix B.

If there are pending invocation in \hat{S} we just add a responding class in which there is a response to each of them and they output all values written in the execution. Observe that \hat{S} respects the real-time order of the invocations and responses of E because if the response of p_i precedes the invocation of p_j then set_i cannot contain p_j and then $\tau_i < \tau_j$, which implies that the invocation of p_j in \hat{S} happens after the invocation of p_i . Thus, the algorithm is interval-linearizable.

□*Theorem 1*

4.3 Interval-linearizability is composable and non-blocking

Even though interval-linearizability is much more general than linearizability it retains some of its benefits. Proofs are in Appendix E.

Theorem 2 (Locality of interval-linearizability). *An execution E is interval-linearizable if and only if $E|_X$ is interval-linearizable, for every object X .*

Proof. We prove that if each $E|_X$ is interval-linearizable for every X , then E is interval-linearizable (the other direction is trivial). Consider an interval-linearization $\hat{S}|_X = (S_X, \xrightarrow{S_X})$ of $E|_X$. Let R_X be the responses appended to E_X to get $\hat{S}|_X$ and let \bar{E} be the extension of E obtained by appending the responses in the sets R_X in some order. Let $\bar{OP} = (OP, \xrightarrow{op})$ be the partial order associated to $comp(\bar{E})$.

We define the following relation $\hat{S} = (S, \xrightarrow{S})$. The set S is the union of all S_X , namely, the union of all concurrency classes in the linearizations of all objects. The relation \xrightarrow{S} is defined as follows:

1. For every object X , $\xrightarrow{S_X} \subseteq \xrightarrow{S}$.
2. For every pair of distinct objects X and Y , for every $a \in OP|_X$ and $b \in OP|_Y$ such that $a \xrightarrow{op} b$ and $a \in S'$ and $b \in S''$, for a responding class $S' \in S$ and an invoking class $S'' \in S$, we define $S' \xrightarrow{S} S''$.

Claim 1. *The relation \xrightarrow{S} is acyclic.*

Although \xrightarrow{S} is acyclic, it might not be transitive. Consider the transitive closure $\xrightarrow{\bar{S}}$ of \xrightarrow{S} . One can easily show that $\xrightarrow{\bar{S}}$ is acyclic, hence it is a partial order over S . It is well-known that a partial order can be extended to a total order. Let $\hat{S}^* = (S, \xrightarrow{S^*})$ a total order obtained from $\xrightarrow{\bar{S}}$. It could be that in \hat{S}^* concurrency classes do not alternate between invoking and responding, however, the first concurrency class certainly is an invoking one. To get an interval-sequential execution, we merge consecutive invoking classes and responding classes in \hat{S}^* (namely, we take the union of such a sequence) and adjust $\xrightarrow{S^*}$ accordingly. Let $\hat{T}^* = (T, \xrightarrow{T^*})$ be the resulting interval-sequential execution. We claim that \hat{T}^* is an interval-sequential linearization of E .

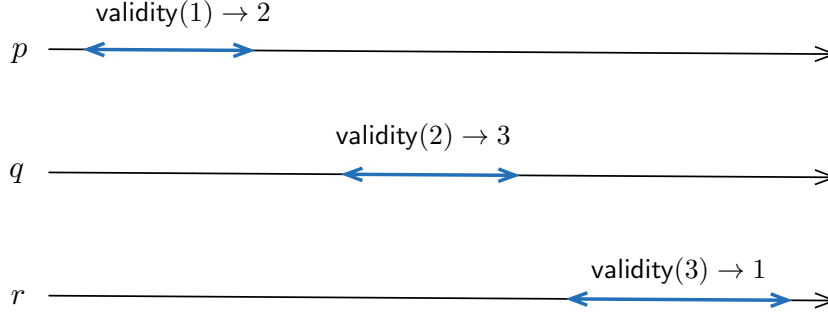


Figure 10: An execution that does not satisfy the *validity* task.

By the definition of $\hat{S} = (S, \xrightarrow{S})$ above, we have that for every object X , $\hat{T}^*|_X \in ISS(X)$. From the assumption that each $\hat{S}|_X$ respects the real time order in $comp(\overline{E}|_X)$, and by the definition of \hat{S} , it follows that $\xrightarrow{T^*}$ respects the real time order in $comp(\overline{E})$, namely, $\xrightarrow{T^*}$ respects \xrightarrow{op} . That and the definition of $\hat{S} = (S, \xrightarrow{S})$ also imply that for every process p , $comp(\overline{E})|_p = \hat{T}_p^*$. This completes the proof of the lemma. \square

When we consider the specification $ISS(X)$ of an interval-sequential object with total operation $opName$, for every $S \in ISS(X)$ and every invocation $\{inv(opName)\}$ to $opName$, the interval-sequential execution $S \cdot \{inv(opName)\} \cdot S'$ belongs to $ISS(X)$, for some responding concurrency class containing a matching response to $\{inv(opName)\}$.

Theorem 3. *Let E be an interval-linearizable execution in which there is a pending invocation $inv(op)$ of a total operation. Then, there is a response $res(op)$ such that $E \cdot res(op)$ is interval-linearizable.*

5 Tasks and their relationship with automata-based specifications

A task is a static way of specifying a *one-shot* concurrent problem, namely, a problem with one operation that can be invoked once by each process. Here we study the relationship between this static way of defining a problem, and the automata-based ways of specifying a problem that we have been considering. Proofs and additional details are in Appendix E.

Roughly, a task $(\mathcal{I}, \mathcal{O}, \Delta)$ consists of a set of input assignments \mathcal{I} , and a set of output assignments \mathcal{O} , which are defined in terms of sets called *simplexes* of the form $s = \{(id_1, x_1), \dots, (id_k, x_k)\}$. A singleton simplex is a *vertex*. A simplex s is used to denote the input values, or output values in an execution, where x_i denotes the value of the process with identity id_i , either an input value, or an output value. Both \mathcal{I} and \mathcal{O} are *complexes*, which means they are closed under containment. There is an input/output relation Δ , specifying for each input simplex $s \in \mathcal{I}$, a subcomplex of \mathcal{O} consisting of a set of output simplexes $\Delta(s) \subseteq \mathcal{O}$ that may be produced with input s . If s, s' are two simplexes in \mathcal{I} with $s' \subset s$, then $\Delta(s') \subset \Delta(s)$. Formal definitions are in Appendix D.

When does an execution satisfy a task? A task is usually specified informally, in the style of Section 2.2. E.g., for the k -set agreement task one would say that each process proposes a value, and decides a value, such that (validity) a decided value has been proposed, and (agreement) at most k different values are decided. A formal definition of when an execution satisfies a task is derived next. A task T has only one operation, $task()$, which process id_i may call with value x_i , if (id_i, x_i) is a vertex of \mathcal{I} . The operation $task(x_i)$ may return y_i to the process, if (id_i, y_i) is a vertex of \mathcal{O} . Let E be an execution where each process calls $task()$ once. Then, σ_E denotes the simplex containing all input vertices in E , namely, if in E there is an invocation of $task(x_i)$ by process id_i then (id_i, x_i) is in σ_E . Similarly, τ_E denotes the simplex containing all output vertices in E , namely, (id_i, y_i) is in τ_E iff there is a response y_i to a process id_i in E . We say that E *satisfies* task $T = \langle \mathcal{I}, \mathcal{O}, \Delta \rangle$ if for every prefix E' of E , it holds that $\tau_{E'} \in \Delta(\sigma_{E'})$. It is necessary to consider all prefixes of an execution, to prevent anomalous executions that globally seem correct, but in a prefix a process predicts future invocations, as in the execution of the *validity* task in Figure 10.³

³This prefix requirement has been implicitly considered in the past by stating that an algorithm solves a task if any of its executions agree with the specification of the task.

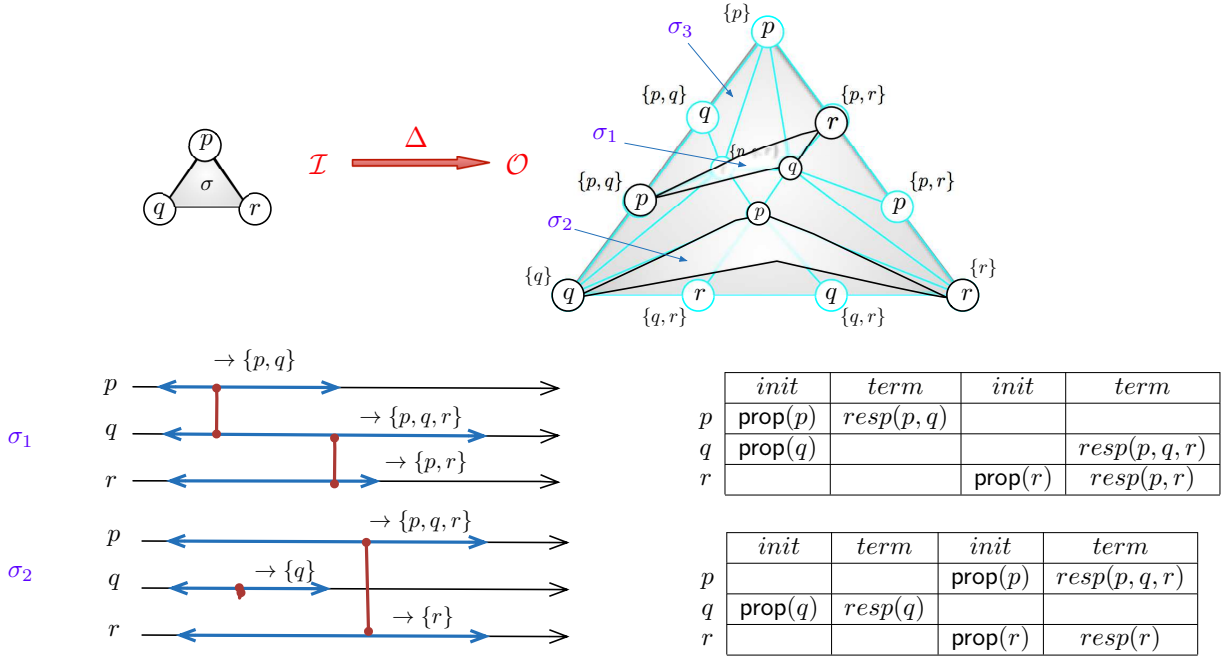


Figure 11: Two special output simplexes σ_1, σ_2 , and interval-linearizations of two executions with corresponding outputs

From tasks to interval-sequential objects A task is a very compact way of specifying a distributed problem that is capable of describing allowed behaviours for certain concurrency patterns, and indeed it is hard to understand what exactly is the problem being specified. The following theorem (with its proof) provides an automata-based representation of a task, explaining which outputs may be produced in each execution, as permitted by Δ .

Theorem 4. *For every task T , there is an interval-sequential object O_T such that an execution E satisfies T if and only if it is interval-linearizable with respect to O_T .*

To give an intuition of the insights in the proofs of this theorem, consider the immediate snapshot task (Figure 16). A simple case is the output simplex σ_4 in the center of the output complex, where the three processes output $\{p, q, r\}$. It is simple, because this simplex does not intersect the boundary. Thus, it can be produced as output only when all three operations are concurrent. More interesting is output simplex σ_3 , where they also may run concurrently, but in addition, the same outputs may be returned in a fully sequential execution, because σ_3 intersects both the 0-dimensional and the 1-dimensional boundary of the output complex. In fact σ_3 can also be produced if p, q are concurrent, and later comes r , because 2 vertices of σ_3 are in $\Delta(p, q)$. Now, consider the two more awkward output simplexes σ_1, σ_2 in $\Delta(\sigma)$ added to the immediate-snapshot output complex in Figure 11, where $\sigma_1 = \{(p, \{p, q\}), (q, \{p, q, r\}), (r, \{p, r\})\}$, and $\sigma_2 = \{(p, \{p, q, r\}), (q, \{q\}), (r, \{r\})\}$. At the bottom of the figure, two executions and their interval-linearizations are shown, though there are more executions that are interval-linearizable and can produce σ_1 and σ_2 . Consider σ_2 , which is in $\Delta(\sigma)$. Simplex σ_2 has a face, $\{q\}$, in $\Delta(q)$, and another face, $\{r\}$ in $\Delta(r)$. This specifies a different behavior from the output simplex in the center, than does not intersect with the boundary. Since $\Delta(\{q\}) = \{q\}$, it is OK for q to return $\{q\}$ when it invokes and returns before the others invoke. Now, since $\{\{p, q, r\}, q, r\} \in \Delta(\{p, q, r\})$ then it is OK for r to return $\{r\}$ after everybody has invoked. Similarly, since $\{\{p, q, r\}, q, r\} \in \Delta(\{p, q, r\})$, p can return $\{p, q, r\}, q, r\}$. The main observation here is that the structure of the mapping Δ encodes the interval-sequential executions that can produce the outputs in a given output simplex. In the example, Δ precludes the possibility that in a sequential execution the processes outputs the values in σ_1 , since Δ specifies no process can decide without seeing anyone else.

From one-shot interval-sequential objects to tasks The converse of Theorem 4 is not true. Lemma 1 shows that even some sequential objects, such as queues, cannot be represented as a task. Also, recall that there are tasks with no set-sequential specification. Thus, both tasks and set-sequential objects are interval-sequential objects, but they are incomparable.

Lemma 1. *There is a sequential one-shot object O such that there is no task T_O , satisfying that an execution E is linearizable with respect to O if and only if E satisfies T_O (for every E).*

We have established that tasks have strictly less expressive power than interval-sequential one-shot objects, however, a slight modification of the notion of tasks allows to equate the power of both approaches for specifying distributed one-shot problems. Roughly speaking, tasks cannot model interval-sequential objects because they do not have a mechanism to encode the state of an object. The extension we propose below allows to model states.

In a *refined* task $T = \langle \mathcal{I}, \mathcal{O}, \Delta \rangle$, \mathcal{I} is defined as usual and each output vertex of \mathcal{O} has the form (id_i, y_i, σ'_i) where id_i and y_i are, as usual, the ID of a process and an output value, and σ'_i is an input simplex called the *set-view* of id_i . The properties of Δ are maintained and in addition it satisfies the following: for every $\sigma \in \mathcal{I}$, for every $(id_i, y_i, \sigma'_i) \in \Delta(\sigma)$, it holds that $\sigma'_i \subseteq \sigma$. An execution E *satisfies* a refined task T if for every prefix E' of E , it holds that $\Delta(\sigma_{E'})$ contains the simplex $\{(id_i, y_i, \sigma_{iE''}) : (id_i, y_i) \in \tau_{E'} \wedge E'' \text{ (which defines } \sigma_{iE''}) \text{ is the shortest prefix of } E' \text{ containing the response } (id_i, y_i))\}$.

We stress that, for each input simplex σ , for each output vertex $(id_i, y_i, \sigma_i) \in \Delta(\sigma)$, σ_i is a way to model distinct output vertexes in $\Delta(\sigma)$ whose output values (in (id_i, y_i)) are the same, then a process that outputs that vertex does not actually output σ_i . In fact, the set-view of a process id_i corresponds to the set of invocations that precede the response (id_i, y_i) to its invocation in a given execution (intuitively, the invocations that a process “sees” while computing its output value). Set-views are the tool to encode the state of an object. Also observe that if E satisfies a refined task T , then the set-views behave like snapshots: 1) a process itself (formally, its invocation) appears in its set-view and 2) all set-view are ordered by containment (since we assume E is well-formed).

As already mentioned, interval-sequential objects and refined tasks have the same ability to specify distributed one-shot problems, as the following theorems show. The proof of Theorem 6 is essentially the same as the proof of Theorem 4.

Theorem 5. *For every one-shot interval-sequential object O with a single total operation, there is a refined task T_O such that any execution E is interval-linearizable with respect to O if and only if E satisfies T_O .*

Theorem 6. *For every refined task T , there is an interval-sequential object O_T such that an execution E satisfies T if and only if it is interval-linearizable with respect to O_T .*

6 Conclusion

We have proposed the notion of an *interval-sequential object*, specified by a state machine similar to the ones used for sequentially specified objects, except that transitions are labeled with sets of invocations and responses, instead of operations, to represent operations that span several consecutive transitions. Thus, in a state an invocation might be pending. The corresponding consistency condition is *interval-linearizability*. If an execution is interval-linearizable for an object X , its invocations and responses can be organized, respecting real-time, in a way that they can be executed through the automaton of X . Thus, contrary to the case of linearizability where to linearize an execution one has to find unique linearization points, for interval-linearizability one needs to identify an interval of time for each operation, and the intervals might overlap. We have shown that by going from linearizability to interval-linearizability one does not sacrifice the properties of being local and non-blocking.

We have discovered that interval-sequential objects have strictly more expressive power than tasks. Any algorithm that solves a given task is interval-linearizable with respect to the interval-sequential object that corresponds to the task, however, there are one-shot objects that cannot be expressed as tasks. We introduced the notion of *refined tasks* and prove that interval-sequential objects and refined tasks are just two different styles, equally expressive, of specifying concurrent one-shot problems, the first operational, and the second static. This brings benefits from each style to the other, and finally provides a common framework to think about linearizability, set-linearizability, interval-linearizability, and tasks.

There are various directions interesting to pursue further. In the domain of concurrent specifications, there is interest in comparing the expressive power of several models of concurrency, e.g. [19], and as far as we know, no model similar to ours has been considered. Higher dimensional automata [39], the most expressive model in [19], seems related to set-linearizability. Also, several papers explore partial order semantics of programs. More flexible notions of linearizability, relating two arbitrary sets of histories appear in [15], but without stating a compositionality result, and without an automata-based formalism. However it is worth exploring this direction further, as it establishes that linearizability implies observational refinement, which usually entails compositionality (see, e.g., [23]). Also, it would be interesting to consider that in this semantics two events in a single trace can be related in three ways: definitely dependent, definitely concurrent or unrelated.

Several versions of non-determinism were explored in [10], which could be understood through the notions in this paper. Also, it would be interesting to consider multi-shot task versions that correspond to interval-sequential objects, as well as the implications of the locality property.

As observed in [24], devising linearizable objects can be very difficult, requiring complex algorithms to work correctly under general circumstances, and often resulting in bad average-case behavior. Programmers thus optimize algorithms to handle common scenarios more efficiently. The authors propose *speculative linearizability* to simplify the design of efficient yet robust linearizable protocols. It would be interesting to see if similar techniques can be used for interval-specifications of concurrent objects proposed here, and if our more generic composability proof sheds light on the composability result of [24].

Often concurrent data structures shared require linear worst case time to perform a single instance of an operation in any non-blocking implementation [14], else, they are not linearizable e.g. [29]. Thus, concurrent specifications, such as interval-linearizable objects open possibilities of sub-linear time implementations.

Finally, Shavit [44] summarizes beautifully the common knowledge state that “it is infinitely easier and more intuitive for us humans to specify how abstract data structures behave in a sequential setting. Thus, the standard approach to arguing the safety properties of a concurrent data structure is to specify the structure’s properties sequentially, and find a way to map its concurrent executions to these ‘correct’ sequential ones.” We hope interval-linearizability opens the possibility of facilitating reasoning about concurrent specifications, when no sequential specifications are appropriate.

Acknowledgments

A. Castañeda was partially supported by a PAPIIT-UNAM research grant. S. Rajsbaum was partially supported by a PAPIIT-UNAM, and a LAISLA Mexico-France research grant. M. Raynal was partially supported by the French ANR project DISPLEXITY devoted to computability and complexity in distributed computing, and the Franco-German ANR project DISCMAT devoted to connections between mathematics and distributed computing.

References

- [1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873-890, 1993.
- [2] Afek Y., Gafni E., and Lieber O., Tight group renaming on groups of size g is equivalent to g -consensus. *Proc. 23th Int'l Symposium on Distributed Computing (DISC'09)*, Springer LNCS 5805, pp. 111-126, 2009.
- [3] Ahamad M., Neiger G., Burns J.E., Hutto P.W., and Kohli P. Causal memory: definitions, implementation and programming. *Distributed Computing*, 9:37-49, 1995.
- [4] Aspnes J. and Ellen F., Tight bounds for adopt-commit objects. *Theory Computing Systems*, 55(3): 451-474 (2014).
- [5] Attiya H. and Rajsbaum S., The combinatorial structure of wait-free solvable tasks. *SIAM Journal of Computing*, 31(4): 1286-1313, 2002.
- [6] Attiya H. and Welch J., *Distributed computing: fundamentals, simulations and advanced topics*, (2d Edition), Wiley-Interscience, 414 pages, 2004.
- [7] Borowsky E. and Gafni E., Immediate atomic snapshots and fast renaming. *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC'93)*, ACM Press, pp. 41-51, 1993.
- [8] Borowsky E., Gafni E., Lynch N. and Rajsbaum S., The BG distributed simulation algorithm. *Distributed Computing* 14(3): 127-146, 2001.
- [9] Chandra T.D., Hadzilacos V., Jayanti P., Toueg S.: Generalized irreducibility of consensus and the equivalence of t -resilient and wait-free implementations of consensus. *SIAM Journal of Computing* 34(2): 333-357, 2004.
- [10] Castañeda A., Rajsbaum S., and Raynal M., Agreement via symmetry breaking: on the structure of weak subconsensus tasks. *Proc. 27th IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS'13)*, IEEE Press, pp. 1147-1158, 2013.
- [11] Chaudhuri S., More choices allow more faults: set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132-158, 1993.
- [12] Conde R., Rajsbaum S., The complexity gap between consensus and safe-consensus (Extended Abstract). *Proc. 21th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'14)*, Springer LNCS 8576, pp. 68-82, 2014.
- [13] Dziuina, D., Fatourou, P., and Kanellou, E., Survey on consistency conditions. *FORTH-ICS TR 439*, December 2013. <https://www.ics.forth.gr/tech-reports2013/2013>.
- [14] Ellen F., Hendler D., and Shavit N., On the inherent sequentiality of concurrent objects. *SIAM Journal of Computing*, 41(3): 519-536, 2012.
- [15] Filipović I., O'Hearn P., Rinetky N., and Yang H., Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51-52):4379-4398, 2010.
- [16] Friedman R., Vitenberg R., and Chokler G., On the composability of consistency conditions. *Information Processing Letters*, 86(4):169-176, 2003.
- [17] Gafni E., Round-by-round fault detectors: unifying synchrony and asynchrony. *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC'98)*, ACM Press, pp. 143-152, 1998.
- [18] Gafni E., Snapshot for time: the one-shot case. *arXiv:1408.3432v1*, 10 pages, 2014.
- [19] van Glabbeek R.J., On the expressiveness of higher dimensional automata. *Theoretical Computer Science*, 356(3):265-290, 2006.
- [20] Gafni E. and Rajsbaum S., Distributed programming with tasks. *Proc. 14th Int'l Conference On Principles Of Distributed Systems (OPODIS'010)*, Springer LNCS 6490, pp. 205-218, 2010.

- [21] Garg V.K. and Raynal M., Normality: a consistency condition for concurrent objects. *Parallel Processing Letters*, 9(1):123–134, 1999.
- [22] Gilbert S. and Golab W., Making sense of relativistic distributed systems. *Proc. 28th Int'l Symposium on Distributed Computing (DISC'14)*, Springer LNCS 8784, pp. 361-375, 2014.
- [23] Gotsman A., Musuvathi M. and Yang H., Show no weakness: sequentially consistent specifications of TSO libraries. *Proc. 26th Int'l Symposium on Distributed Computing (DISC'12)*, Springer LNCS 7611, pp. 31–45, 2012.
- [24] Guerraoui R., Kuncak V., and Losa G., Speculative linearizability. *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI'12)*, ACM Press, pp. 55-66, 2012.
- [25] Hemed N. and Rinetzky N.? Brief Announcement: Concurrency-aware linearizability. *Proc. 33th ACM Symposium on Principles of Distributed Computing (PODC'14)*, page 209–211, ACM Press, 2014.
- [26] Herlihy M., Kozlov D., and Rajsbaum S., *Distributed computing through combinatorial topology*, Morgan Kaufmann/Elsevier, 336 pages, 2014 (ISBN 9780124045781).
- [27] Herlihy M., Rajsbaum S., Raynal M., Power and limits of distributed computing shared memory models. *Theoretical Computer Science*, 509: 3-24, 2013.
- [28] Herlihy M. and Shavit N., *The art of multiprocessor programming*. Morgan Kaufmann, 508 pages, 2008 (ISBN 978-0-12-370591-4).
- [29] Herlihy M., Shavit N., and Waarts O., Linearizable counting networks. *Distributed Computing* 9(4): 193-203 (1996).
- [30] Herlihy M. and Wing J., Axioms for concurrent objects. *Proc. 14th ACM Symposium on Principles of Programming Languages (POPL'87)*, ACM Press, pp. 13-26, 1987.
- [31] Herlihy M. and Wing J., Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
- [32] Keleher P., Cox A.L., and Zwaenepoel W., Lazy release consistency for software distributed shared memory. *Proc. 19th ACM Int'l Symposium on Computer Architecture (ISCA'92)*, pages 13–21, 1992.
- [33] Lamport L., How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C28(9):690–691, 1979.
- [34] Lamport L., On inter-process communications, Part I: basic formalism, Part II: algorithms. *Distributed Computing*, 1(2):77–101, 1986.
- [35] Li K. and Hudak P., Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.
- [36] Misra J., Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142-153, 1986.
- [37] Moran S. and Wolfstahl Y., Extended impossibility results for asynchronous complete networks, *Information Processing Letters*, Volume 26, Issue 3, 23 November 1987, Pages 145–151.
- [38] Neiger G., Set-linearizability. Brief announcement in *Proc. 13th ACM Symposium on Principles of Distributed Computing (PODC'94)*, ACM Press, page 396, 1994.
- [39] Pratt, V.R., Modeling concurrency with geometry. *Proc. 18th Annual ACM Symposium on Principles of Programming Languages (POPL'91)*, ACM Press, pp. 311–322, 1991.
- [40] Rajsbaum S., Raynal M., and Travers C., An impossibility about failure detectors in the iterated immediate snapshot model. *Information Processing Letters*, 108(3): 160-164, 2008.
- [41] Raynal M. and Schiper A., A suite of definitions for consistency criteria in distributed shared memories, *Annales des Télécommunications*, 52:11-12, 1997.
- [42] Raynal M., *Concurrent programming: algorithms, principles, and foundations*. Springer, 530 pages, 2013 (ISBN 978-3-642-32026-2).
- [43] Raynal M., *Distributed algorithms for message-passing systems*. Springer, 510 pages, 2013 (ISBN: 978-3-642-38122-5).
- [44] Shavit N., Data structures in the multicore age. *Communications of the ACM*, 54(3):76-84, 2011.
- [45] Saks M.E. and Zaharoglou F., Wait-free k-set agreement is impossible: the topology of public knowledge. *SIAM Journal of Computing*, 29(5):1449-1483, 2000.
- [46] Taubenfeld G., *Synchronization algorithms and concurrent programming*. Pearson Education/Prentice Hall, 423 pages, 2006 (ISBN 0-131-97259-6).
- [47] Vogels W., Eventually consistent. *Communications of the ACM*, 52(1):40-44, 2009.

A Linearizability

A *sequential object* O is a (not necessarily finite) Mealy state machine (Q, Inv, Res, δ) whose output values are determined both by its current state $s \in Q$ and the current input $I \in Inv$. If O is in state q and it receives as input an invocation $in \in Inv$ by process p , then, if $\delta(q, in) = (r, q')$, the meaning is that O may return the response r to the invocation in by process p , and move to state q' . Notice that the response r has to be to the invocation by p , but there may be several possible responses (if the object is non-deterministic). Also, it is convenient to require that the object is *total*, meaning that for any state q , $\delta(q, I) \neq \emptyset$, for all $I \in Inv$.

Considering any object defined by a sequential specification on total operations, linearizability [31] generalizes the notion of an atomic read/write object formalized in [34, 36], and encountered in virtual memory-based distributed systems [35].

Intuitively, an execution is linearizable if it could have been produced by multiplexing the processes on a single processor. This definition considers complete histories. If the execution is partial, an associated complete execution can be defined as follows. The local execution $\hat{H}|i$ of each process p_i for which the last operation is pending (i.e., p_i issued an invocation and there no matching response event), is completed with a response matching the invocation event. Thus, it may be possible to associate different complete histories with a given partial execution.

An execution E is linearizable if there is an extension \bar{E} of E and a sequential execution \hat{S} such that:

- $comp(\bar{E})$ and \hat{S} are equivalent (no process can distinguish between $comp(\bar{E})$ and \hat{S}).
- \hat{S} is legal (the specification of each object is respected).
- The total order \hat{S} respects the partial order \widehat{OP} associated to $comp(\bar{E})$ (any two operations ordered in \widehat{OP} are ordered the same way in \hat{S}).

As shown in [31], the linearizability consistency condition has the “composability” property (called “locality” in [31]), which states that a computation E is linearizable if and only if, for each of its objects X , $E|X$ is linearizable.

B Additional details about the write-snapshot task

Recall that in the write-snapshot task the $write()$ and $snapshot()$ operations are merged to define a single operation denoted $write_snapshot()$. It satisfies the self-inclusion and containment properties. Notice that the *immediate snapshot* task [7] which motivated Neiger to propose set-linearizability [38] is a write-snapshot which additionally satisfies the following immediacy property: $\forall i, j : [(\langle j, - \rangle set_i) \wedge (\langle i, - \rangle set_j)] \Rightarrow (set_i = set_j)$.

For completeness and comparison, we include the following proof in the usual, somewhat informal style, of the correctness of the write-snapshot algorithm. To simplify the presentation we suppose that the value written by p_i is i , and the pair $\langle i, v_i \rangle$ is consequently denoted i .

Theorem 7. *The algorithm of Figure 2 wait-free implements write-snapshot.*

Proof Let us first show that the invocation of $write_snapshot()$ by any process p_i terminates. As there is a bounded number of processes, and a process invokes $write_snapshot()$ at most once, it follows that a process can be forced to execute at most $(n - 1)$ double collects, and the termination follows.

The self-inclusion property follows immediately from line 01, and the fact that no value is ever withdrawn from the array MEM .

To prove the containment property, let us consider two processes p_i and p_j , which return set_i and set_j , respectively. Let us first consider p_i . As it returns set_i , we have $set_i = old_i = new_i$ where new_i corresponds to the last asynchronous read of $MEM[1..n]$ by p_i , and old_i corresponds to the previous asynchronous read of $MEM[1..n]$. Let $\tau[old_i]$ the time at which terminates the read of $MEM[1..n]$ returning old_i , and $\tau[new_i]$ the time at which starts the read of $MEM[1..n]$ returning new_i . As $old_i = new_i$, it follows that there is a time τ_i , such that $\tau[old_i] \leq \tau_i \leq \tau[new_i]$ and, due to the termination predicate of line 05, the set of non- \perp values of $MEM[1..n]$ at time τ_i is equal to set_i .

The same applies to p_j , and there is consequently a time τ_j at which the set of non- \perp values of $MEM[1..n]$ is equal to set_j . Finally, as (1) $\tau_i \leq \tau_j$ or $\tau_i > \tau_j$, and (b) values written in $MEM[1..n]$ are never withdrawn, it follows that we necessarily have $set_i \subseteq set_j$ or $set_j \subseteq set_i$. $\square_{Theorem 7}$

A finite state automaton describing the behavior of a write-snapshot object The non-deterministic automaton of Figure 12 describes in an abbreviated form all the possible behaviors of a write-snapshot object in a system of three processes p , q , and r . To simplify the figure, it is assumed that a process p_i proposes i . Each edge correspond to an invocation of `write_snapshot()`, and the list of integers L labeling a transition edge means that the corresponding invocation of `write_snapshot()` is by one of the processes p_i such that $i \in L$. The value returned by the object is $\{L\}$. Thus, for the linearization of the execution in Figure 3, the path in the automaton goes through states \emptyset , $\{1, 2\}$, $\{1, 2\}$, $\{1, 2, 3\}$.

Any path starting from the initial empty state, and in which a process index appears at most once, defines an execution of the write-snapshot task that does not predict the future. Moreover if, when it executes, a process proceeds from the automaton state s_1 to the state s_2 , the state s_2 defines the tuple of values output by its invocation of `write_snapshot()`.

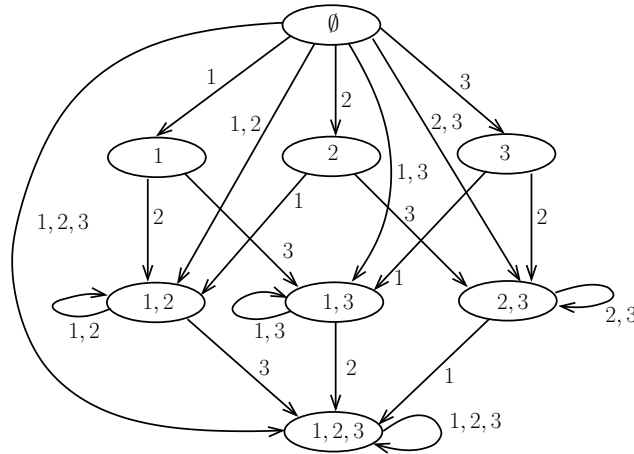


Figure 12: A non-deterministic automaton for a write-snapshot object

C Additional discussion and examples of linearizability limitations

C.1 Additional examples of tasks with no sequential specification

Several tasks have been identified that are problematic for dealing with them through linearizability. The problem is that they do not have a natural sequential specification. One may consider linearizable implementations of restricted sequential specifications, where if two operations occur concurrently, one is linearized before the other. Thus, in every execution, always there is a first operation. In all cases we discuss below, such an implementation would provably be of a more powerful object.

An *adopt-commit* object [17] is a one-shot shared-memory object useful to implement round-based protocols for set-agreement and consensus. It supports a single operation, `adopt_commit()`. The result of this operation is an output of the form $(commit, v)$ or $(adopt, v)$, where the second component is a value from this set and the 1st component indicates whether the process should decide value v immediately or adopt it as its preferred value in later rounds of the protocol. It has been shown to be equivalent to the *conflict detection* object [4], which supports a single operation, `check()`. It returns true or false, and has the following two properties: In any execution that contains a `check(v)` operation and a `check(v')` operation with $v \neq v'$, at least one of these operations returns true. In any execution in which all `check` operations have the same input value, they all return false. As observed in [4] neither *adopt-commit* objects nor *conflict detectors* have sequential specification. A deterministic linearizable implementation of an *adopt-commit* object gives rise to a deterministic implementation of consensus, which does not exist. Similarly, the first `check` operation linearized in any execution of a *conflict detector* must return false and subsequent `check` operations with different inputs must return true, which can be used to implement test-and-set, for which no deterministic implementation from registers exists.

In the *safe-consensus* problem of [2], the agreement condition of consensus is retained, but the validity condition is weakened as follows: if the first process to invoke it returns before any other process invokes it, then it outputs its input; otherwise the consensus output can be arbitrary, not even the input of any process. There is no sequential specification of this problem, because in any sequential specification, the first process to be linearized would obtain its own proposed value. See Appendix C.3.

Two examples that motivated Neiger are the following [38]. In the *immediate snapshot* task [7], there is a single operation `Immediate_snapshot()`, such that a snapshot occurs immediately after a read. Such executions play an important role in distributed computability [5, 7, 45]. There is no sequential specification of this task. One may consider linearizable implementations of restricted immediate snapshot behavior, where if two operations occur concurrently, one is linearized before the other, and where the first operation does not return the value by the second. But such an implementation would provably be of a more powerful object (immediate snapshots can be implemented wait-free using only read/write registers), that could simulate test-and-set.

The other prominent example exhibited in [38] is the *k-set agreement* task [11], where processes agree on at most k of their input values. Any linearizable implementation restricts the behavior of the specification, because some process final value would have to be its own input value. This would be an artifact imposed by linearizability. Moreover, there are implementations of set agreement with executions where no process chooses its own initial value.

C.2 Splitting operations to model concurrency

One is tempted to separate an operation into two, an invocation and a response, to specify the effect of concurrent invocations. Consider two operations of an object, $op_1()$ and $op_2()$, such that each one is invoked with a parameter and can return a value. Suppose we want to specify how the object behaves when both are invoked concurrently. We can separate each one into two operations, $inv_op_i()$ and $resp_op_i()$. When a process wants to invoke $op_i(x)$, instead it first invokes $inv_op_i(x)$, and once the operation terminates, it invokes $resp_op_i()$, to get back the output parameter. Then a sequential specification can define what the operation returns when the history is $inv_op_1(x_1), inv_op_2(x_2), resp_op_1(), resp_op_2()$.

k -Set agreement is easily transformed into an object with a sequential specification, simply by accessing it through two different operations, one that deposits a value into the object and another that returns one of the values in the object. Using a non-deterministic specification that remembers which values the object has received so far, and which ones have so far been returned, one captures the behavior that at most k values are returned, and any of the proposed values can be returned. This trick can be used in any task.

Separating an operation into a proposal operation and a returning operation has several problems. First, the program is forced to produce two operations, and wait for two responses. There is a consequent loss of clarity in the code of the program, in addition to a loss in performance, incurred by a two-round trip delay. Also, the intended meaning of linearization points is lost; an operation is now linearized at *two* linearization points. Furthermore, the resulting object may provably *not* be the same. A phenomenon that has been observed several times (see, e.g., in [12, 20, 40]) is that the power of the object can be increased, if one is allowed to invoke another object in between the two operations. Consider a test-and-set object that returns either 0 or 1, and the write-snapshot object. It is possible to solve consensus among 2 processes with *only one* snapshot object and one test-and-set object only if it is allowed to invoke test-and-set in between the write and the snapshot operation. Similarly, consider a safe-consensus object instead of the test-and-set object. If one is allowed to invoke in between the two operations of write-snapshot a safe-consensus object, then one can solve consensus more efficiently [12].

The object corresponding to a task with two operations Let T be a task $(\mathcal{I}, \mathcal{O}, \Delta)$. We will model T as a sequential object O_T in which each process can invoke two operations, **set** and **get**, in that order. The idea is that **set** communicates to O_T the input value of a process, while **get** produces an output value to a process. Thus, the unique operation of T is modelled with two operations. The resulting sequential object is non-deterministic.

We define O_T . The set of invocations and responses are the following:

$$Inv(O_T) = \{\text{set}(p_i, in_i) \mid (p_i, in_i) \in \mathcal{I}\} \cup \{\text{get}(p_i) \mid p_i \in \Pi\}$$

$$Res(O_T) = \{\text{set}(p_i, in_i) : OK \mid (p_i, in_i) \in \mathcal{I}\} \cup \{\text{get}(p_i) : out_i \mid p_i \in \Pi \wedge (p_i, out_i) \in \mathcal{O}\}$$

The set of states of O_T is $Q = \{(\sigma, \tau) \mid \sigma \in \mathcal{I} \wedge \tau \in \Delta(\sigma)\}$. Intuitively, a set (σ, τ) represents that the inputs and output O_T knows at that state are σ and τ . The initial state of is (\emptyset, \emptyset) . We define δ as follows. Let (σ, τ) and (σ', τ') be two states of O_T . Then,

- If $\tau = \tau', \sigma \neq \sigma'$ and $\sigma' = \{\sigma \cup (p_i, in_i)\} \in \mathcal{I}$, then $\delta((\sigma, \tau), \text{set}(p_i, in_i))$ contains the tuple $(\text{set}(p_i, in_i) : OK, (\sigma', \tau'))$.
- If $\sigma = \sigma', \tau \neq \tau'$ and $\tau' = \{\tau \cup (p_i, out_i)\} \in \Delta(\sigma)$, then $\delta((\sigma, \tau), \text{get}(p_i))$ contains the tuple $(\text{get}(p_i) : out_i, (\sigma', \tau'))$.

Note that for every sequential execution \widehat{S} of O_T , it holds that $\tau_{\widehat{S}} \in \Delta(\sigma_{\widehat{S}})$, where $\sigma_{\widehat{S}}$ is the input simplex containing every input vertex in \widehat{S} and, similarly, $\tau_{\widehat{S}}$ is the output simplex containing every output simplex in \widehat{S} .

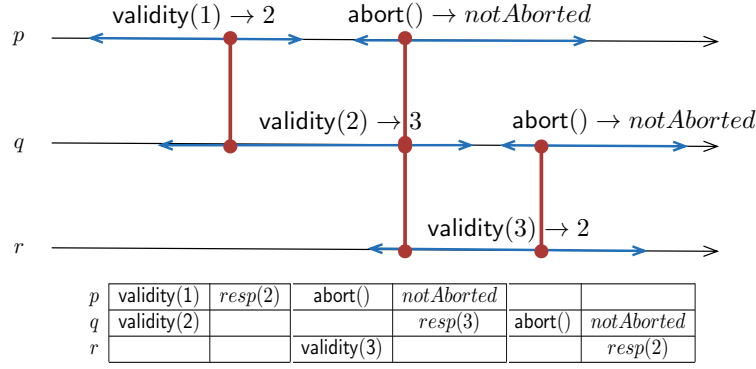


Figure 13: An execution of a Validity-Abort object (3)

C.3 Validity and Safe-consensus objects

We first discuss the validity object with abort, and then the safe-consensus object.

C.3.1 Validity with abort object

An interval-sequential object can be enriched with an abort operation that takes effect only if a given number of processes request an abort concurrently. Here we describe the example of Section 3.3 in more detail, that extends the validity object with an abort operation that should be invoked concurrently by at least k processes. As soon as at least k processes concurrently invoke abort the object will return from then on aborted to every operation. Whenever less than k processes are concurrently invoking abort, the object may return NotAborted to any pending abort. An example appeared in Figure 6, for $k = 2$. Another example is in Figure 13, where it is shown that even though there are two concurrent abort operations, they do not take effect because they are not observed concurrently by the object. This illustrates why this paper is only about safety properties, the concepts here cannot enforce liveness. There is no way of guaranteeing that the object will abort even in an execution where all processes issue abort at the same time, because the operations may be executed sequentially.

The k -*validity-abort* object is formally specified as an interval-sequential object by an automaton, that can be invoked by either propose(v) or abort, and it responds with either resp(v) or aborted or NotAborted. Each state q is labeled with three values: $q.vals$ is the set of values that have been proposed so far, $q.pend$ is the set of processes with pending invocations, and $q.aborts$ is the set of processes with pending abort. The initial state q_0 has $q_0.vals = \emptyset$, $q_0.pend = \emptyset$ and $q_0.aborts = \emptyset$. If in is an invocation to the object different from abort, let $val(in)$ be the proposed value, and if r is a response from the object, let $val(r)$ be the responded value.

For a set of invocations I (resp. responses R) $vals(I)$ denotes the proposed values in I (resp. $vals(R)$). Also, $aborts(I)$ denotes the set of processes issuing an reqAbort in I , and $notAborted(R)$ is the set of processes getting notAborted in R .

The transition relation $\delta(q, I)$ contains all pairs (R, q') such that:

1. If $r \in R$ then $id(r) \in q.pend$ or there is an $in \in I$ with $id(in) = id(r)$,
2. If $(r = \text{resp}(v) \in R \text{ or } \text{notAborted} \in R)$ then $\text{aborted} \notin R$,
3. If $r = \text{resp}(v) \in R$ then $val(r) = v \in q.vals$ or there is an $in \in I$ with $val(in) = val(r)$,
4. If $\text{notAborted} \in R$, then $0 < |q.aborts| + |aborts(I)| < k$
5. If $|q.aborts| + |aborts(I)| \geq k$ then $\text{aborted} \in R$.
6. $q'.vals = q.val \cup vals(I)$, $q'.pend = (q.pend \cup ids(I)) \setminus ids(R)$, and $q'.aborts = (q.aborts \cup aborts(I)) \setminus notAborted(R)$

C.3.2 Safe-consensus

Recall that the *safe-consensus* problem of [2], is similar to consensus. The agreement condition of consensus is retained, but the validity condition is weakened as follows: if the first process to invoke it returns before any other

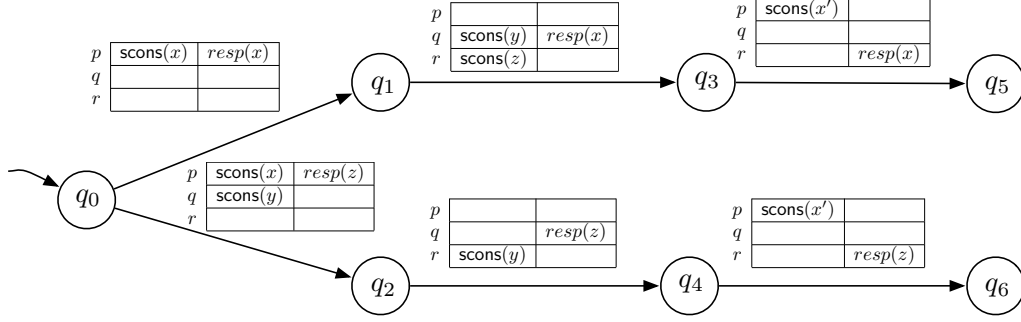


Figure 14: Part of an interval-sequential automaton of safe-consensus

process invokes it, then it outputs its input; otherwise the consensus output can be arbitrary, not even the input of any process. As noticed in Section C.1, there is no sequential specification of this problem.

See Figure 14 for part of the automata corresponding to safe-consensus, and examples of interval executions in Figure 15.

Interval execution α_1						
	<i>init</i>	<i>term</i>	<i>init</i>	<i>term</i>	<i>init</i>	<i>term</i>
<i>p</i>	scons (<i>x</i>)	<i>resp</i> (<i>x</i>)			scons (<i>x'</i>)	
<i>q</i>			scons (<i>y</i>)	<i>resp</i> (<i>x</i>)		
<i>r</i>			scons (<i>z</i>)			<i>resp</i> (<i>x</i>)

Interval execution α_2						
	<i>init</i>	<i>term</i>	<i>init</i>	<i>term</i>	<i>init</i>	<i>term</i>
<i>p</i>	scons (<i>x</i>)	<i>resp</i> (<i>z</i>)			scons (<i>x'</i>)	
<i>q</i>	scons (<i>y</i>)			<i>resp</i> (<i>z</i>)		
<i>r</i>			scons (<i>z</i>)			<i>resp</i> (<i>z</i>)

Figure 15: Examples of interval-executions for safe-consensus

D Tasks

D.1 Basic definitions

A task is the basic distributed equivalent of a function, defined by a set of inputs to the processes and for each (distributed) input to the processes, a set of legal (distributed) outputs of the processes, e.g., [26]. In an algorithm designed to solve a task, each process starts with a private input value and has to eventually decide irrevocably on an output value. A process p_i is initially not aware of the inputs of other processes. Consider an execution where only a subset of k processes participate; the others crash without taking any steps. A set of pairs $s = \{(\text{id}_1, x_1), \dots, (\text{id}_k, x_k)\}$ is used to denote the input values, or output values, in the execution, where x_i denotes the value of the process with identity id_i , either an input value, or an output value.

A set s as above is called a *simplex*, and if the values are input values, it is an *input simplex*, if they are output values, it is an *output simplex*. The elements of s are called *vertices*. An *input vertex* $v = (\text{id}_i, x_i)$ represents the initial state of process id_i , while an *output vertex* represents its decision. The *dimension* of a simplex s is $|s| - 1$, and it is *full* if it contains n vertices, one for each process. A subset of a simplex is called a *face*. Since any number of processes may crash, simplexes of all dimensions are of interest, for taking into account executions where only processes in the simplex participate. Therefore, the set of possible input simplexes forms a *complex* because its sets are closed under containment. Similarly, the set of possible output simplexes also form a complex.

More generally, a *complex* \mathcal{K} is a set of vertices $V(\mathcal{K})$, and a family of finite, nonempty subsets of $V(\mathcal{K})$, called *simplexes*, satisfying: (1) if $v \in V(\mathcal{K})$ then $\{v\}$ is a simplex, and (2) if s is a simplex, so is every nonempty subset of s . The dimension of \mathcal{K} is the largest dimension of its simplexes, and \mathcal{K} is *pure* of dimension k if every

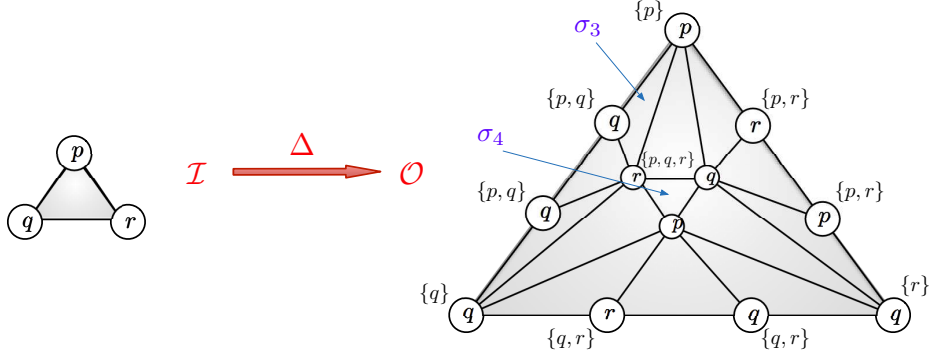


Figure 16: Immediate snapshot task

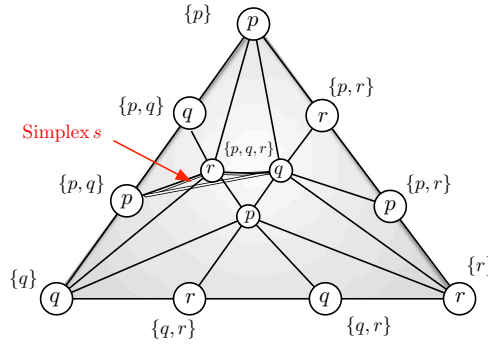


Figure 17: Part of the write-snapshot output complex

simplex belongs to a k -dimensional simplex. In distributed computing, the simplexes (and complexes) are often *chromatic*, since each vertex v of a simplex is labeled with a distinct process identity.

Definition 2 (Task). A task T for n processes is a triple $(\mathcal{I}, \mathcal{O}, \Delta)$ where \mathcal{I} and \mathcal{O} are pure chromatic $(n - 1)$ -dimensional complexes, and Δ maps each simplex s from \mathcal{I} to a subcomplex $\Delta(s)$ of \mathcal{O} , satisfying:

1. $\Delta(s)$ is pure of dimension s
2. For every t in $\Delta(s)$ of dimension s , $\text{ID}(t) = \text{ID}(s)$
3. If s, s' are two simplexes in \mathcal{I} with $s' \subset s$ then $\Delta(s') \subset \Delta(s)$.

We say that Δ is a *carrier map* from the input complex \mathcal{I} to the output complex \mathcal{O} .

A task is a very compact way of specifying a distributed problem, and indeed it is hard to understand what exactly is the problem being specified. Intuitively, Δ specifies, for every simplex $s \in \mathcal{I}$, the valid outputs $\Delta(s)$ for the processes in $\text{ID}(s)$ assuming they run to completion, and the other processes crash initially, and do not take any steps.

The immediate snapshot task is depicted in Figure 16. On the left, the input simplex is depicted and, on the right, the output complex appears.

In figure 17 one simplex s is added to the output complex of the immediate snapshot task of Figure 16, where $s = \{(p, \{p, q\}), (q, \{p, q, r\}), (r, \{p, q, r\})\}$. This simplex s corresponds to the execution of Figure 4.

D.2 Validity as a task

Recall the validity object is specified as an interval-sequential object in Section 3.3, which is neither linearizable nor set-linearizable. In the usual, informal style of specifying a task, the definition would be very simple: an operation returns a value that has been proposed. A bit more formally, in an execution where a set of processes participate with inputs I (each $x \in I$ is proposed by at least one process), each participating process decides a value in I . To illustrate why this informal style can be misleading, consider the execution in Figure 10, where the

three processes propose values $I = \{1, 2, 3\}$, so according to the informal description it should be ok that they decide values $\{1, 2, 3\}$. However, for the detailed interleaving of the figure, it is not possible that p and q would have produced outputs that they have not yet seen.

To define validity formally as a task, the following notation will be useful. It defines a complex that represents all possible assignments of (not necessarily distinct) values from a set U to the processes. In particular, all processes can get the same value x , for any $x \in U$. Given any finite set U and any integer $n \geq 1$, we denote by $\text{complex}(U, n)$ the $(n - 1)$ -dimensional *pseudosphere* [26] complex induced by U : for each $i \in [n]$ and each $x \in U$, there is a vertex labeled (i, x) in the vertex set of $\text{complex}(U, n)$. Moreover, $u = \{(\text{id}_1, u_1), \dots, (\text{id}_k, u_k)\}$ is a simplex of $\text{complex}(U, n)$ if and only if u is properly colored with identities, that is $\text{id}_i \neq \text{id}_j$ for every $1 \leq i < j \leq k$. In particular, $\text{complex}(\{0, 1\}, n)$ is (topologically equivalent) to the $(n - 1)$ -dimensional sphere. For $u \in \text{complex}(U, n)$, we denote by $\text{val}(u)$ the set formed of all the values in U corresponding to the processes in u . Similarly, for any set of processes P , $\text{complex}(U, P)$ is the $|P| - 1$ -dimensional pseudosphere where each vertex is labeled with a process in P , and gets a value from U .

The *validity task* over a set of values U that can be proposed, is $(\mathcal{I}, \mathcal{O}, \Delta)$, where $\mathcal{I} = \mathcal{O} = \text{complex}(U, n)$. The carrier map Δ is defined as follows. For each simplex $s \in \mathcal{I}$, $\Delta(s) = \text{complex}(U', P')$, where P' is the set of processes appearing in s and U' is their proposed values.

E Proofs

Claim 1 The relation \xrightarrow{S} is acyclic.

Proof For the sake of contradiction, suppose that \xrightarrow{S} is not acyclic, namely, there is a cycle $C = S_1 \xrightarrow{S} S_2 \xrightarrow{S} \dots \xrightarrow{S} S_{m-1} \xrightarrow{S} S_m$, with $S_1 = S_m$. We will show that the existence of C implies that $\xrightarrow{S_X}$ is not acyclic, for some object X , which is a contradiction to our initial assumptions.

First note that it cannot be that each S_i is a concurrency class of the same object X , because if so then C is a cycle of $\xrightarrow{S_X}$, which contradicts that $\xrightarrow{S_X}$ is a total order. Thus, in C there are concurrency classes of several objects.

In what follows, by slight abuse of notation, we will write $S' \xrightarrow{op} S''$ if S' and S'' are related in \hat{S} because of the second case in the definition of \xrightarrow{S} .

Note that in C there is no sequence $S_1 \xrightarrow{op} S_2 \xrightarrow{op} S_3$ because in \hat{S} whenever $T' \xrightarrow{op} T''$, we have that T' is a responding class and T'' is an invoking class, by definition. Thus, in C there must be a sequence $S_1 \xrightarrow{op} S_2 \xrightarrow{S_X} \dots \xrightarrow{S_X} S_t \xrightarrow{op} S_{t+1} \xrightarrow{S_Y} S_{t+2}$. Observe that in \hat{S} , we have $S_2 \xrightarrow{S_X} S_t$ since $\xrightarrow{S_X}$ is transitive, hence the sequence can be shortened: $S_1 \xrightarrow{op} S_2 \xrightarrow{S_X} S_t \xrightarrow{op} S_{t+1} \xrightarrow{S_Y} S_{t+2}$. Note that S_1 and S_t are responding classes while S_2 and S_{t+1} are invoking classes.

Now, since $S_1 \xrightarrow{op} S_2$, there are operations $a, b \in OP$ such that $a \xrightarrow{op} b$, $a \in S_1$ and $b \in S_2$. Similarly, for $S_t \xrightarrow{op} S_{t+1}$, there are $c, d \in OP$ such that $c \xrightarrow{op} d$, $c \in S_t$ and $d \in S_{t+1}$. This implies that $\text{term}(a) < \text{init}(b)$ and $\text{term}(c) < \text{init}(d)$. Observe that if we show $a \xrightarrow{op} d$ then, by definition of \hat{S} , we have $S_1 \xrightarrow{op} S_{t+1}$, if S_1 and S_{t+1} are concurrent classes of distinct objects, and $S_1 \xrightarrow{S_Y} S_{t+1}$ otherwise. Hence, the sequence $S_1 \xrightarrow{op} S_2 \xrightarrow{S_X} S_t \xrightarrow{op} S_{t+1} \xrightarrow{S_Y} S_{t+2}$ can be shortened to $S_1 \xrightarrow{op} S_{t+1} \xrightarrow{S_Y} S_{t+2}$ or $S_1 \xrightarrow{S_Y} S_{t+1} \xrightarrow{S_Y} S_{t+2}$. Repeating this enough times, in the end we get that there are concurrency classes S_i, S_j in C such that $S_i \xrightarrow{S_X} S_j \xrightarrow{S_X} S_i$, which is a contradiction since $\xrightarrow{S_X}$ is acyclic, by hypothesis.

To complete the proof of the claim, we need to show that $a \xrightarrow{op} d$, i.e., $\text{term}(a) < \text{init}(d)$. We have four cases:

1. if $b = c$, then $\text{term}(a) < \text{init}(b) < \text{term}(b) = \text{term}(c) < \text{init}(d)$, hence $a \xrightarrow{op} d$.
2. If $b \xrightarrow{op} c$, then $\text{term}(a) < \text{init}(b) < \text{term}(b) < \text{init}(c) < \text{term}(c) < \text{init}(d)$, hence $a \xrightarrow{op} d$.
3. If $c \xrightarrow{op} b$, then we have that $S_t \xrightarrow{S_X} S_2$, because each $\hat{S}|_X = (S_X, \xrightarrow{S_X})$ respects the real time order in $\text{comp}(\overline{E}|_X)$, by hypothesis. But we also have that $S_2 \xrightarrow{S_X} S_t$, which contradicts that $\xrightarrow{S_X}$ is a total order. Thus this case cannot happen.
4. If b and c are concurrent, i.e., $b \not\xrightarrow{op} c$ and $c \not\xrightarrow{op} b$, then note that if $\text{init}(d) \leq \text{term}(a)$, then $\text{term}(c) < \text{init}(d) \leq \text{term}(a) < \text{init}(b)$, which implies that $c \xrightarrow{op} b$ and hence b and c are not concurrent, from which follows that $\text{term}(a) < \text{init}(d)$.

□_{Claim 1}

Theorem 3 Let E be an interval-linearizable execution in which there is a pending invocation $inv(op)$ of a total operation. Then, there is a response $res(op)$ such that $E \cdot res(op)$ is interval-linearizable.

Proof Since E is interval-linearizable, there is an interval-linearization $\hat{S} \in ISS(X)$ of it. If $inv(op)$ appears in \hat{S} , we are done, because \hat{S} contains only completed operations and actually it is an interval-linearization of $E \cdot res(op)$, where $res(op)$ is the response to $inv(op)$ in \hat{S} .

Otherwise, since the operation is total, there is a responding concurrency class S' such that $\hat{S} \cdot \{inv(op)\} \cdot S' \in ISS(X)$, which is an interval-linearization of $E \cdot res(op)$, where $res(op)$ is the response in S' matching $inv(op)$.

□_{Theorem 3}

```

function sequences( $E$ ) is
   $i \leftarrow 1$ ;  $e \leftarrow$  first event in  $E$ ;  $F \leftarrow$  empty execution
   $\sigma_0, \tau_0 \leftarrow \emptyset$ ;  $A \leftarrow \sigma_0$ ;  $B \leftarrow \tau_0$ ;
  while  $F \neq E$  do
    if ( $e$  is an invocation)
      then if ( $\tau_i \setminus \tau_{i-1} = \emptyset$ )
        then  $\sigma_i \leftarrow \sigma_i \cup \{e\}$ 
        else  $\sigma_{i+1} \leftarrow \sigma_i \cup \{e\}$ ;  $\tau_{i+1} \leftarrow \tau_i$ ;
           $A \leftarrow A \cdot \sigma_i$ ;  $B \leftarrow B \cdot \tau_i$ ;  $i \leftarrow i + 1$ 
        end if
      else  $\tau_i \leftarrow \tau_i \cup \{e\}$ 
    end if;
     $F \leftarrow F \cdot e$ ;  $e \leftarrow$  next event to  $e$  in  $E$ 
  end while;
   $A \leftarrow A \cdot \sigma_i$ ;  $B \leftarrow B \cdot \tau_i$ ; return ( $A, B$ ).

```

Figure 18: Producing a sequence of faces of the invocation simplex σ and response simplex τ of an execution E .

For the rest of the section we will often use the following notation. Let E be an execution. Then, σ_E and τ_E denote the sets containing all invocations and responses of E , respectively.

Claim 2. For every execution E , the function sequences() in Figure 18 produces sequences $A = \sigma_0, \dots, \sigma_k$ and $B = \tau_0, \dots, \tau_m$ such that

1. $k = m$.
2. $\sigma_0 = \emptyset \subset \sigma_1 \subset \dots \subset \sigma_{m-1} \subset \sigma_m = \sigma_E$ and $\tau_0 = \emptyset \subset \tau_1 \subset \dots \subset \tau_{m-1} \subseteq \tau_m = \tau_E$.
3. If E has no pending invocations, then $\tau_{m-1} \subset \tau_m$, otherwise $\tau_{m-1} = \tau_m$.
4. If E satisfies a task with carrier map Δ , then, that for each i , $\tau_i \in \Delta(\sigma_i)$.
5. For every response e and invocation e' in E such that e precedes e' and they do not match each other, we have $i < j$, where i is the smallest integer such that $e \in \tau_i$ and j is the smallest integer such that $e' \in \sigma_j$.
6. For every response e of E in $\tau_i \setminus \tau_{i-1}$, σ_i contains all invocations preceding e in E .

Proof Items (1), (2) and (6) follow directly from the code. For item (3), note that if E has no pending invocations, it necessarily ends with a response, which is added to τ_m , and thus $\tau_{m-1} \subset \tau_m$. For item (4), consider a pair σ_i and τ_i , and let E' be the shortest prefix of E that contains each response in τ_i . Note that the simplex containing all invocations in E' is σ_i . Since, by hypothesis, E satisfy T , it follows that $\tau_i \in \Delta(\sigma_i)$.

For item (5), consider such events e and e' . Since e precedes e' , the procedure analyzes first e , from which follows that it is necessarily true that $i \leq j$, so we just need to prove that $i \neq j$. Suppose, by contradiction, that $i = j$. Consider the beginning of the while loop when e' is analyzed. Note that $e' \notin \sigma_i$ at that moment. Also, note that $e \in \tau_i \setminus \tau_{i-1}$ because i is the smallest integer such that $e \in \tau_i$ and its was analyzed before e' . Thus, when the procedure process e' , puts it in σ_{i+1} , which is a contradiction, because in the final sequence of simplexes $e' \notin \sigma_i$.

□_{Claim 2}

Theorem 4 For every task T , there is an interval-sequential object O_T such that any execution E satisfies T if and only if it is interval-linearizable with respect to O_T .

Proof The structure of the proof is the following. (1) First, we define O_T using T , (2) then, we show that every execution that satisfies T , is interval-linearizable with respect to O_T , and (3) finally, we prove that every execution that is interval-linearizable with respect to O_T , satisfies T .

Defining O_T : Let $T = \langle \mathcal{I}, \mathcal{O}, \Delta \rangle$. To define O_T , we first define its sets with invocations, response and states: $Inv = \{(id, x) : \{(id, x)\} \in \mathcal{I}\}$, $Res = \{(id, y) : \{(id, y)\} \in \mathcal{O}\}$ and $Q = \{(\sigma, \tau) : \sigma \in \mathcal{I} \wedge \tau \in \mathcal{O}\}$. The interval-sequential object O_T has one initial state: (\emptyset, \emptyset) . Then O_T will have only one operation and so the name of it does not appear in the invocation and responses.

The transition function δ is defined as follows. Consider an input simplex σ of T and let E be an execution that satisfies T with $\sigma_E = \sigma$ and $\tau_E \in \Delta(\sigma_E)$. Consider the sequences $\sigma_0 = \emptyset \subset \sigma_1 \subset \dots \subset \sigma_m = \sigma_E$ and $\tau_0 = \emptyset \subset \tau_1 \subset \dots \subset \tau_m = \tau_E$ that **Sequences** in Figure 18 produces on E . Then, for every $i = 1, \dots, m$, $\delta((\sigma_{i-1}, \tau_{i-1}), \sigma_i \setminus \sigma_{i-1})$ contains $((\sigma_i, \tau_i), \tau_i \setminus \tau_{i-1})$. In other words, we use the sequences of faces to define an interval-sequential execution (informally, a grid) that will be accepted by O_T : the execution has $2m$ concurrency classes, and for each $i = 1, \dots, m$, the invocation $(p_j, -)$ (of process p_j) belongs to the $2i - 1$ -th concurrency class if $(p_j, -) \in \sigma_i \setminus \sigma_{i-1}$, and the response to the invocation of appears in the $2i$ -th concurrency class if $(p_j, -) \in \tau_i \setminus \tau_{i-1}$. We repeat the previous construction for every such σ and E .

If E satisfies T , it is interval-linearizable: Consider an execution E that satisfies T . We prove that E is interval-linearizable with respect to O_T . Since E satisfies T , we have that $\tau_E \in \Delta(\sigma_E)$. By definition, $\Delta(\sigma)$ is $\dim(\sigma)$ -dimensional pure, then there is a $\dim(\sigma)$ -dimensional $\gamma \in \Delta(\sigma)$ such that τ_E is a face of γ . Let \overline{E} be an extension of E in which the responses in $\gamma \setminus \tau_E$ are added in some order. Thus, there are no pending operation in \overline{E} . Consider the sequences of simplexes produced by **Sequences** in Figure 18 on \overline{E} . As we did when defined O_T , the two sequence define an interval-sequential execution \hat{S} . We have the following: (1) \hat{S} is an interval-sequential execution of O_T , by construction, (2) for every p , $\hat{S}|_p = \overline{E}|_p$, by construction, and (3) Claim 2.5 implies that \hat{S} respect the real-time order of invocations and responses in \overline{E} : if $op_1 \xrightarrow{op} op_2$ in the partial order $\hat{O}P = (OP, \xrightarrow{op})$ associated to \overline{E} , then, by the claim, the response of op_1 appears for the first time the sequence in τ_i and the invocation of op_2 appears for the first in the sequence in σ_j , with $i < j$, and hence, by construction, op_1 precedes op_2 in \hat{S} . We conclude that \hat{S} is an interval-linearization of E .

If E is interval-linearizable, it satisfies T : Consider an execution E that is interval-linearizable with respect to O_T . We will show that E satisfies T . There is an interval-sequential execution \hat{S} that is a linearization of E , since E is interval-linearizable. Consider any prefix E' of E and let \hat{S}' be the shortest prefix of \hat{S} such that (1) it is an interval-sequential execution and (2) every completed invocation in E' is completed in \hat{S}' (note that there might be pending invocations in E' that does not appear in \hat{S}'). By construction, \hat{S}' defines two sequences of simplexes $\sigma_0 = \emptyset \subset \sigma_1 \subset \dots \subset \sigma_m$ and $\tau_0 = \emptyset \subset \tau_1 \subset \dots \subset \tau_m$ with $\tau_i \in \Delta(\sigma_i)$, for every $i = 1, \dots, m$. Observe that $\sigma_{E'} = \sigma_m$ and $\tau_{E'} \subseteq \tau_m$, and thus $\tau_{E'} \in \Delta(\sigma_{E'})$ because $\tau_m \in \Delta(\sigma_m)$. What we have proved holds for every prefix E' of E , then we conclude that E satisfies T . $\square_{Theorem 4}$

Lemma 1 There is a sequential one-shot object O such that there is no task T_O , satisfying that an execution E is linearizable with respect to O if and only if E satisfies T_O (for every E).

Proof Consider a restricted queue O for three processes, p , q and r , in which, in every execution, p and q invoke $enq(1)$ and $enq(2)$, respectively, and r invokes $deq()$. If the queue is empty, r 's dequeue operation gets \perp .

Suppose, for the sake of contradiction, that there is a corresponding task $T_O = (\mathcal{I}, \mathcal{O}, \Delta)$, as required by the lemma. The input complex \mathcal{I} consists of one vertex for each possible operation by a process, namely, the set of vertices is $\{(p, enq(1)), (q, enq(2)), (r, deq())\}$, and \mathcal{I} consists of all subsets of this set. Similarly, the output complex \mathcal{O} contains one vertex for every possible response to a process, so it consists of the set of vertices $\{(p, ok), (q, ok), (r, 1), (r, 2), (r, \perp)\}$. It should contain a simplex $\sigma_x = \{(p, ok), (q, ok), (r, x)\}$ for each value of $x \in \{1, 2, \perp\}$, because there are executions where p, q, r get such values, respectively. See Figure 19.

Now, consider the three sequential executions of the figure, α_1, α_2 and α_\perp . In α_1 the process execute their operations in the order p, q, r , while in α_2 the order is q, p, r . In α_1 the response to r is 1, and if α_2 it is 2. Given that these executions are linearizable for O , they should be valid for T_O . This means that every prefix of α_1 should be valid:

$$\begin{aligned} \{(p, ok)\} &= \Delta((p, enq(1))) \\ \{(p, ok), (q, ok)\} &\in \Delta(\{(p, enq(1)), (q, enq(2))\}) \\ \sigma_1 = \{(p, ok), (q, ok), (r, 1)\} &\in \Delta(\{(p, enq(1)), (q, enq(2)), (r, deq())\}) = \Delta(\sigma) \end{aligned}$$

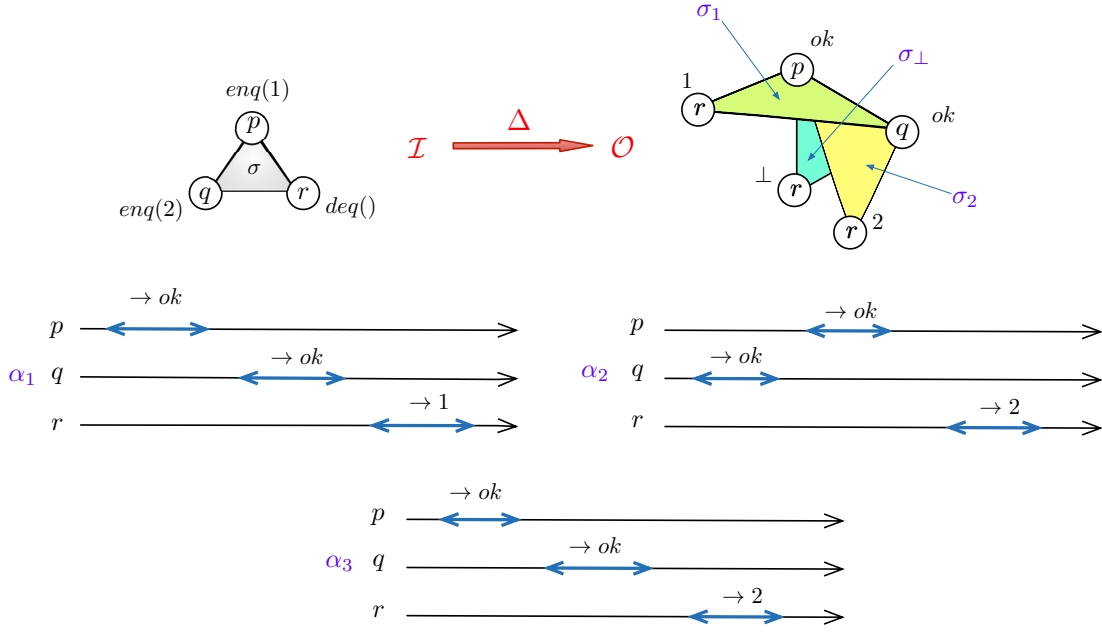


Figure 19: Counterexample for a simple queue object

Similarly from α_2 we get that

$$\sigma_2 = \{(p, ok), (q, ok), (r, 2)\} \in \Delta(\sigma)$$

But now consider α_3 , with the same sequential order p, q, r of operations, but now r gets back value 2. This execution is not linearizable for O , but is accepted by T_O because each of the prefixes of α_3 is valid. More precisely, the set of inputs and the set of outputs of α_2 are identical to the sets of inputs and set of outputs of α_3 .

□_{Lemma 1}

Theorem 5 For every one-shot interval-sequential object O with a single total operation, there is a refined task T_O such that any execution E is interval-linearizable with respect to O if and only if E satisfies T_O .

Proof The structure of the proof is the following. (1) First, we define T_O using O , (2) then, we show that every execution that is interval-linearizable with respect to O , satisfies T_O , and (3) finally, we prove that every execution that satisfies T_O , is interval-linearizable with respect to O .

Defining T_O : We define the refined task $T_O = (\mathcal{I}, \mathcal{O}, \Delta)$. First, since O has only one operation, we assume that its invocation and responses have the form $inv(p_j, x_j)$ and $res(p_j, y_j)$. Let Inv and Res be the sets with the invocations and responses of O . Each subset σ of Inv containing invocations with different processes, is a simplex of \mathcal{I} . It is not hard to see that \mathcal{I} is a chromatic pure $(n - 1)$ -dimensional complex. \mathcal{O} is defined similarly: each subset τ of $Res \times 2^{Inv}$ containing responses in the first entry with distinct processes, is a simplex of \mathcal{O} .

We often use the following construction in the rest of the proof. Let E be an execution. Recall that σ_E and τ_E are the simplexes (sets) containing all invocations and responses in E . Let $\tau_0 = \emptyset \subset \tau_1 \subset \dots \subset \tau_m = \tau_E$ and $\sigma_0 = \emptyset \subset \sigma_1 \subset \dots \subset \sigma_m = \sigma_E$ be the sequences of simplexes produced by **Sequences** in Figure 18 on E . These sequences define an output simplex $\{(res, \sigma_i) : res \in \tau_i \setminus \tau_{i-1}\}$ of T_O , which will be denoted γ_E .

We define Δ as follows. Consider an input simplex $\sigma \in \mathcal{I}$. Suppose that E is an execution such that (1) $\sigma_E = \sigma$ (2) it has no pending invocations and (3) it is interval-linearizable with respect to O . Note that $\dim(\sigma) = \dim(\tau_E)$ and $ID(\sigma) = ID(\tau_E)$. Consider the simplex γ_E induced by E , as defined above. Note that $\dim(\sigma) = \dim(\gamma_E)$ and $ID(\sigma) = ID(\gamma_E)$. Then, $\Delta(\sigma)$ contains γ_E and all its faces. We define Δ by repeating this for each such σ and E .

Before proving that T is a task, we observe the following. Every $\dim(\sigma)$ -simplex $\gamma_E \in \Delta(\sigma)$ is induced by an execution E that is interval-linearizable. Let \hat{S} be an interval-linearization of E . Then, for any execution F such that $\gamma_F = \gamma_E$ (namely, **Sequences** produce the same sequences on E and on F), \hat{S} is an interval-linearization of F : (1) by definition, \hat{S} is an interval-sequential execution of O , (2) for every process p , $F|_p = \hat{S}|_p$, and (3) \hat{S}

respects the real-time order of invocations and response in F because it respects that order in E and, as already said, they have the same sequence of simplexes and, by Claim 2.5, these sequences reflect the real-time order of invocations and responses.

We argue that T is a refined task. Clearly, for each $\sigma \in \mathcal{I}$, $\Delta(\sigma)$ is a pure and chromatic complex of dimension $\dim(\sigma)$, and for each $\dim(\sigma)$ -dimensional $\tau \in \Delta(\sigma)$, $ID(\tau) = ID(\sigma)$. Consider now a proper face σ' of σ . By definition, each $\dim(\sigma')$ -dimensional simplex $\gamma_{E'} \in \Delta(\sigma')$ corresponds to an execution E' whose set of invocations is σ' , has no pending invocations and is interval-linearizable with respect to O . Let \hat{S}' be an interval-linearization of E' and let s' be the state O reaches after running \hat{S}' . Since the operation of O is total, from the state s' , the invocations in $\sigma \setminus \sigma'$ can be invoked one by one until O reaches a state s in which all invocations in σ have a matching response. Let \hat{S} be the corresponding interval-sequential execution and let E be an extension of E' in which (1) every invocation in $\sigma \setminus \sigma'$ has the response in \hat{S} and (2) all invocations first are appended in some order and then all responses are appended in some (not necessarily the same) order. Note that \hat{S}' is a prefix of \hat{S} and actually \hat{S} is an interval-linearization of E . Then, $\Delta(\sigma)$ contain the $\dim(\sigma)$ -simplex γ_E induced by E . Observe that $\gamma_{E'} \subset \gamma_E$, hence, $\Delta(\sigma') \subset \Delta(\sigma)$. Finally, for every $\dim(\sigma)$ -simplex $\gamma_E \in \Delta(\sigma)$, for every vertex (id_i, y_i, σ_i) of γ_E , we have that $\sigma_i \subseteq \sigma$ because γ_E is defined through an interval-linearizable execution E . Therefore, we conclude that T_O is a task.

If E is interval-linearizable, it satisfies T_O : Consider an execution E that is interval-linearizable with respect to O . We prove that E satisfies T_O . Since E is interval-linearizable, there is an interval-linearization \hat{S} of it. Let E' be any prefix of E and let \hat{S}' be the shortest prefix of \hat{S} such that (1) it is an interval-sequential execution and (2) every completed invocation in E' is completed in \hat{S}' (note that there might be pending invocations in E' that does not appear in \hat{S}'). Note that \hat{S}' is an interval-linearization of E' . Since interval-linearizability is non-blocking, Theorem 3, there is an interval-linearization \hat{S}'' of E' in which every invocation in E' has a response. Let E'' be an extension of E' that contains every response in \hat{S}'' (all missing responses are appended at the end in some order). Note that $\sigma_{E'} = \sigma_{E''}$ and $\tau_{E'} \subseteq \tau_{E''}$. Consider the output simplexes $\gamma_{E'}$ and $\gamma_{E''}$ induced by E' and E'' . Observe that $\gamma_{E'} \subseteq \gamma_{E''}$ (because E' differs from E'' in some responses at the end). By definition of T_O , $\gamma_{E''} \in \Delta(\sigma_{E'})$, and hence $\gamma_{E'} \in \Delta(\sigma_{E'})$. This holds for every prefix E' of E , and thus E satisfies T_O .

If E satisfies T_O , it is interval-linearizable: Consider an execution E that satisfies T_O . We prove that E is interval-linearizable with respect to O . Let γ_E be the output simplex induced by E . Since E satisfies T_O , $\gamma_E \in \Delta(\sigma_E)$. Since $\Delta(\sigma_E)$ is a pure $\dim(\sigma_E)$ -dimensional complex, there is a $\dim(\sigma)$ -simplex $\gamma_{\bar{E}} \in \Delta(\sigma_E)$ such that $\gamma_E \subseteq \gamma_{\bar{E}}$. Observe that $\sigma_E = \sigma_{\bar{E}}$ while $\tau_E \subseteq \tau_{\bar{E}}$. Therefore, \bar{E} is an extension of E in which the responses in $\tau_{\bar{E}} \setminus \tau_E$ are appended in some order at the end of E . Now, by construction, $\gamma_{\bar{E}} \in \Delta(\sigma_E)$ because there is an interval-sequential execution \hat{S} of O that is an interval-linearization of \bar{E} . We have that \hat{S} is an interval-linearization of E as well because, as already mentioned, \bar{E} is an extension of E in which the responses in $\tau_{\bar{E}} \setminus \tau_E$ are appended in some order at the end.

□ Lemma 5